VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Modul RBM a DBM pro program Modeler neuronových sítí

Module RBM and DBM for Program Neural Net Modeler

2020 Bc. Patrik Lyčka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Patrik Lyčka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Modul RBM a DBM pro program Modeler neuronových sítí
Module RBM and DBM for Program Neural Net Modeler

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je rozšířit existující program "Modeler Neuronových Sítí" o modul umožňující vytvářet a učit sítě typu Restricted Boltzmann Machine (RBM) a Deep Bolztmann Machine (DBM).

Body zadání:
1. Nastudovat a popsat problematiku neuronových sítí, Restricted Boltzmann Machine (RBM) a Deep Bolztmann Machine (DBM).
2. Implementace modulu RBM a DBM do programu "Modeler Neuronových Sítí".
3. Nastudovat a popsat problematiku paralelizace RBM a DBM.
4. Paralelizace RBM a DBM pro běh na výpočetních uzlech.
5. Experimenty a testování na zvolených datových sadách.

Práce bude obsahovat:
1. Přehled použitých technologií.
2. Implementaci výše popsané funkcionality.
3. Dokumentaci programového řešení s využitím diagramů jazyka UML.

Seznam doporučené odborné literatury:

[1] ROJAS, Raul. Neural networks: a systematic introduction. New York: Springer-Verlag, c1996. ISBN 3540605053.
[2] GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE. Deep learning. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN 9780262035613.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2019

Datum odevzdání: 30.04.2020

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2020 .....

Rád bych na tomto místě poděkoval mému vedoucímu práce Ing. Davidovi Ježkovi Ph.D. za velmi cenné rady a ochotu kdykoliv pomoci. Dále bych chtěl také poděkovat mé rodině a přítelkyni za jejich trpělivost a podporu v mém studiu.

Abstrakt

Tato diplomová práce se zaměřuje na rozšíření programu Modeler neuronových sítí o sítě Restricted Boltzmann Machine a Deep Boltzmann Machine. Dále se zaměřuje na paralelizaci učení těchto sítí. V této práci jsou popsány uvedené neuronové sítě a možnosti monitorování jejich učení. Jsou zde také popsány možnosti paralelizace neuronových sítí pro distribuovaný výpočet. V rámci této práce byl rozšířen program Modeler neuronových sítí a vytvořen program pro paralelizaci sítí Restricted Boltzmann Machine a Deep Boltzmann Machine využívající datový paralelismus. S využitím těchto programů a superpočítače byla porovnána rychlost učení a úspěšnost klasifikace vzorů mezi sekvenční a paralelní implementací uvedených neuronových sítí na různých datasetech.

Klíčová slova: neuronové sítě, Restricted Boltzmann Machine, Deep Boltzmann Machine, datový paralelismus, distribuovaný výpočet

Abstract

This master thesis focuses on extending the program Neural net modeler with Restricted Boltzmann Machine and Deep Boltzmann Machine networks. Furthermore, this thesis deals with parallelization of learning of these networks. Stated neural networks and monitoring possibilities of their learning are described in this work. Possibilities of neural networks parallelization for distributed computing are also described there. In the scope of this study, the program Neural net modeler was extended and a new program was created, which deals with parallelization of Restricted Boltzmann Machine and Deep Boltzmann Machine networks using data parallelism. Using these programs and a supercomputer, the speed of learning and the success rate of pattern classification were compared between sequential and parallel implementation of the mentioned neural networks on various datasets.

Keywords: neural networks, Restricted Boltzmann Machine, Deep Boltzmann Machine, data parallelism, distributed computing

Obsah

Seznam použitých zkratek a symbolů **9**

Seznam obrázků **10**

Seznam tabulek **11**

Seznam výpisů zdrojového kódu **12**

1 Úvod **13**

2 Neuronové sítě **14**

3 Restricted Boltzmann Machine **16**
3.1 Parametry sítě 16
3.2 Fáze učení 17
3.3 Fáze vybavování 18
3.4 Rozšíření RBM pro klasifikaci 18

4 Deep Boltzmann Machine **20**
4.1 Parametry sítě 20
4.2 Fáze učení 21
4.3 Fáze vybavení a použití DBM pro klasifikaci 23

5 Monitorování učení neuronové sítě **24**
5.1 Pravděpodobnost aktivace skrytých neuronů 24
5.2 Histogramy vah 24
5.3 Filtry 25
5.4 Chyba 25

6 Paralelizace **27**
6.1 Strategie pro paralelizaci neuronové sítě 27
6.2 Strategie pro synchronizaci vah 30
6.3 Komunikační protokoly 33

7 Implementace **35**
7.1 Použité technologie a knihovny 35
7.2 Použité datasety 38
7.3 Program Modeler neuronových sítí 39
7.4 Implementace rozšíření programu 40
7.5 Implementace paralelizace 45

8 Uživatelská příručka **51**
8.1 Modeler neuronových sítí 51
8.2 Program pro distribuovaný výpočet 53

9 Testování a experimenty **54**
9.1 Testování sekvenčního učení 54
9.2 Testování paralelního učení 57

10 Závěr **63**

Literatura **64**

Seznam použitých zkratek a symbolů

CD	–	Contrastive Divergence
CSV	–	Comma-Separated Values
DBM	–	Deep Boltzmann Machine
HPC	–	High-Performance Computing
IP	–	Internet Protocol
JSON	–	JavaScript Object Notation
PS	–	Parameter Server
RBM	–	Restricted Boltzmann Machine
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
UML	–	Unified Modeling Language
XML	–	eXtensible Markup Language

Seznam obrázků

1 Ukázka struktury RBM 16
2 RBM pro klasifikaci 19
3 Ukázka struktury DBM 20
4 Princip zdvojování při předtréninku DBM 21
5 Vizualizace pravděpodobnosti aktivace skrytých neuronů 24
6 Ukázka histogramu vah 25
7 Ukázka filtru 25
8 Ukázka grafu vývoje chyby 26
9 Datový paralelismus 28
10 Vliv velikosti dávky na chybu a výkon 28
11 Modelový paralelismus 29
12 Zřetězené zpracování 29
13 Centralizovaný synchronní přístup 31
14 Centralizovaný asynchronní přístup 32
15 Decentralizovaný synchronní přístup 33
16 Knihovna Aeron 36
17 Ukázka číslic 3 datasetu OPT 38
18 Ukázka číslic 3 datasetu MNIST 38
19 Ukázka znaků a datasetu OCR 39
20 Ukázka siluet datasetu CalTech 101 Silhouettes 39
21 UML třídní diagram první verze implementace neuronové sítě RBM 41
22 UML třídní diagram implementace nástrojů 43
23 UML třídní diagram implementace programu pro paralelizaci 46
24 UML stavový diagram pro uzel typu Slave 47
25 UML aktivitní diagram znázorňující automatický mód 50
26 Ukázka nástroje pro práci s neuronovou sítí 51
27 Ukázka nástroje pro práci s datasety 52

Seznam tabulek

1 Porovnání rychlosti procesu učení různých implementací sítě RBM 42
2 Porovnání velikosti zprávy při různých způsobech serializace 47
3 Konfigurace serverového testovacího prostředí 54
4 Vliv počtu epoch na rychlost a úspěšnost učení sítě RBM 55
5 Vliv počtu skrytých neuronů na rychlost a úspěšnost učení sítě RBM 55
6 Sekvenční testování učení sítě RBM na různých datasetech 56
7 Parametry neuronové sítě DBM 56
8 Sekvenční testování učení sítě DBM na různých datasetech 57
9 Konfigurace uzlu superpočítače Salomon 57
10 Vliv velikosti dávky na rychlost a úspěšnost učení sítě RBM 58
11 Vliv počtu instancí Slave na rychlost a úspěšnost učení sítě RBM 59
12 Vliv počtu epoch v rámci jedné dávky na rychlost a úspěšnost učení sítě RBM . 59
13 Vliv velikosti dávky na rychlost a úspěšnost učení sítě RBM - dataset MNIST . . 60
14 Vliv velikosti dávky na rychlost a úspěšnost učení sítě DBM 61
15 Vliv počtu instancí Slave na rychlost a úspěšnost učení sítě DBM 61

Seznam výpisů zdrojového kódu

1 Ukázka použití knihovny JFreeChart 36
2 Příjemce v knihovně Aeron 37
3 Odesilatel v knihovně Aeron 37
4 Módy spouštění programu pro paralelizaci 48
5 Skript startProgram 57
6 Skript startBatch 58
7 Příkaz pro přidání požadavku na výpočet do fronty 58

1 Úvod

Neuronové sítě [15, 21] jsou algoritmy, které si berou za vzor činnost lidského mozku, jenž je tvořen velkým množstvím vzájemně propletených buněk, nazývaných neurony. Neurony spolu komunikují pomocí elektrických impulzů. Tyto sítě jsou schopny adaptovat se (učit se), což dává neuronovým sítím široké možnosti využití. Pomocí těchto sítí poté můžeme řešit spoustu různých úloh, od rozpoznávání vzorů [11] a tváří [18] až po predikci vývoje časových řad [14]. Neuronových sítí existuje velké množství typů. V rámci této diplomové práce jsem se zabýval sítěmi Restricted Boltzmann Machine [16, 22] a Deep Boltzmann Machine [23, 24, 22]. Nevýhodou neuronových sítí je, že jejich učení obvykle vyžaduje velký výpočetní výkon a trvá velmi dlouhou dobu, stejně jako to je u lidského mozku.

Jednou z možností, jak velkého výkonu a kratší doby učení dosáhnout je pomocí využití paralelizace učení [10]. Na paralelismus můžeme nahlížet ze dvou pohledů, a to jako na paralelismus v rámci jednoho stroje a paralelismus na více strojích. V rámci této diplomové práce jsem se zabýval paralelizací na více strojích (uzlech). Pro paralelizaci neuronových sítí existuje několik různých strategií, jako je třeba datový nebo modelový paralelismus. Klíčové při této paralelizaci je dosáhnout dostatečného zrychlení při přijatelné ztrátě úspěšnosti rozpoznávání neuronové sítě.

Ve druhé kapitole si stručně popíšeme neuronové sítě. Řekneme si obecné principy na kterých fungují, jaké typy sítí například existují a jaké problémy můžeme pomocí neuronových sítí řešit. V kapitole číslo 3 se zaměříme na síť Restricted Boltzmann Machine. Vysvětlíme si, jak tato neuronová síť funguje a jaké má vstupní parametry. V další kapitole se zaměříme na síť Deep Boltzmann Machine. Opět si vysvětlíme, jak tato neuronová síť funguje a jaké má vstupní parametry. V páté kapitole si řekneme něco o možnostech monitorování průběhu učení neuronových sítí, které jsem také naimplementoval do programu Modeler neuronových sítí. V kapitole číslo 6 se podíváme na paralelizaci. Popíšeme si existující strategie paralelizace a komunikační protokoly a vybereme strategii a komunikační protokol pro implementační část. V následující kapitole si ukážeme, jaké technologie byly použity při implementaci, s jakými datasety umí program pracovat a jak je rozšíření programu a program pro paralelní výpočet naimplementován. V osmé kapitole si popíšeme, jak vytvořené programy spustit a jak je používat. V kapitole číslo 9 se podíváme na sekvenční a paralelní experimenty, kdy paralelní experimenty byly prováděny na superpočítači. Ukážeme si, jakých výsledků dosahuje zvolená strategie paralelizace pro tyto neuronové sítě. V poslední kapitole na závěr zhodnotíme výsledky experimentů.

V této práci jsou použity anglické názvy některých neuronových sítí, parametrů a dalších pojmů z důvodu komplikovaného překladu.

2 Neuronové sítě

Neuronové sítě patří do skupiny biologicky inspirovaných výpočtů [19], což jsou výpočty inspirované biologickými jevy, přičemž se využívá poznatků z biologie, informatiky a matematiky. Tato oblast často úzce souvisí s oblastí umělé inteligence a strojového učení. Patří zde mnoho různých algoritmů, jako například:

- Evoluční algoritmy
- Celulární automaty
- Umělé imunitní systémy
- Lindenmayerovy systémy
- Neuronové sítě

Neuronové sítě [15, 21] jsou inspirovány lidským mozkem. Neuronová síť je algoritmus, který si bere za vzor činnost lidského mozku. Mozek je tvořen velkým množstvím vzájemně propletených buněk, nazývaných neurony, jež spolu komunikují pomocí elektrických impulzů. Nicméně současné neuronové sítě zdaleka nedosahují počtu neuronů a propojení jako lidský mozek.

Neuron je základní stavební jednotka neuronové sítě. Jednotlivé neurony jsou vzájemně propojeny spoji. Neuron se všemi vstupními informacemi provede nějakou jednoduchou operaci (například součet) a výsledek pošle prostřednictvím propojení do dalších neuronů. Různé neuronové sítě se od sebe odlišují tím, jak konkrétně jednotlivé neurony fungují a jak jsou mezi sebou propojeny. Každé propojení je ohodnoceno vahami. Takovéto propojení a schopnost tyto váhy adaptovat (učit se), dává neuronové síti široké možnosti využití. Pomocí neuronových sítí poté můžeme řešit spoustu různých úloh, od rozpoznávání vzorů [11] a tváří [18] až po predikci vývoje časových řad [14].

Učení je důležitá součást funkce neuronové sítě. Konkrétní úlohu, kterou má daná neuronová síť vyřešit, již není třeba algoritmizovat. Síti stačí pouze předložit dostatečně velký vzorek dat, popisující řešenou problematiku a síť se sama naučí daný problém řešit. Existují dva druhy učení neuronových sítí - učení s učitelem a učení bez učitele. Učení s učitelem probíhá tak, že máme množinu vstupů a požadovaných odpovědí (výstupů). Vstupy se pustí do sítě, výstup se porovná s požadovaným výstupem a následně se provede korekce vah u propojení tak, aby byl rozdíl mezi skutečným a požadovaným výstupem co nejmenší. Síť se tedy učí ze svých chyb. V případě učení bez učitele známe vstupy, ale již ne požadovaný výstup. Takový typ sítě má většinou za úkol provést rozškatulkování zadané množiny vstupů do konečného počtu tříd na základě podobnosti.

Hlavní předností neuronové sítě je schopnost naučit se odpověď na předložené vstupy a následně u nových vstupů se obracet na svou paměť a odhadovat odpověď. Ukládání, zpracování a předávání informace probíhá prostřednictvím celé neuronové sítě spíše než pomocí určitých paměťových míst. Paměť a zpracování informace [25] je tedy spíše globální, než lokální. Znalosti

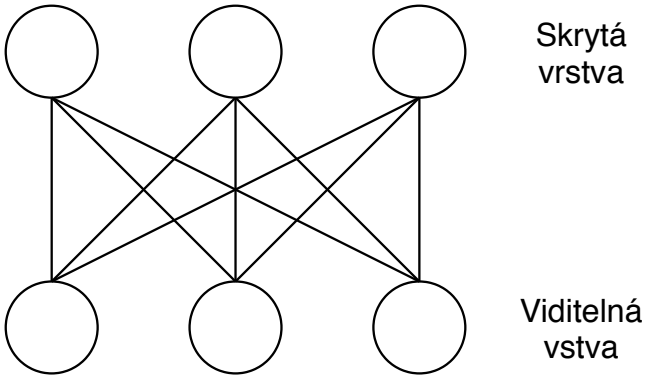
jsou ukládány především prostřednictvím vazeb (propojení) mezi jednotlivými neurony. Vazby mezi neurony, které vedou ke správné interpretaci vstupu, jsou tedy posilovány a naopak vazby, které vedou k nesprávné interpretaci vstupu jsou potlačovány.

Mezi zástupce neuronových sítí patří například:

- Hopfieldova síť
- Kohonenova síť
- Konvoluční síť
- Vícevrstvá síť s algoritmem Backpropagation
- Restricted Boltzmann Machine
- Deep Boltzmann Machine

3 Restricted Boltzmann Machine

Restricted Boltzmann Machine (RBM) [16, 22] je síť se dvěma vrstvami. První vrstva se nazývá viditelná a druhá skrytá. Síť se skládá z množiny N viditelných a M skrytých neuronů. Každý neuron z viditelné vrstvy je propojen s každým neuronem ze skryté vrstvy. Neurony v rámci stejné vrstvy mezi sebou propojeny nejsou. Každý neuron má navíc ještě svůj bias. Struktura této sítě je zobrazena na obrázku 1. Všechny váhy sítě jsou symetrické, tedy $w_{ij} = w_{ji}$ a stav viditelného neuronu v_i i skrytého neuronu h_i v případě binárního modelu může nabývat pouze hodnot $\{0, 1\}$. Hodnota w_{ij} reprezentuje sílu vazby mezi neurony i a j.



Obrázek 1: Ukázka struktury RBM

Každá konfigurace neuronů má svou energii, která je dána rovnicí 1, kde v_i a h_j jsou binární stavy viditelného neuronu i a skrytého neuronu j, a_i a b_j jsou jejich biasy a w_{ij} je váha spoje mezi těmito neurony.

$$E(v, h) = -\sum_{i=1}^{N} a_i v_i - \sum_{j=1}^{M} b_j h_j - \sum_{i=1}^{N} \sum_{j=1}^{M} v_i h_j w_{ij} \tag{1}$$

3.1 Parametry sítě

RBM se může učit pomocí algoritmu Contrastive divergence (CD), který vyžaduje několik vstupních parametrů:

- Počet viditelných neuronů - odpovídá dimenzi vstupních vektorů, určuje počet neuronů ve viditelné vrstvě (celé číslo větší než 0)
- Počet skrytých neuronů - určuje počet neuronů ve skryté vrstvě (celé číslo větší než 0)
- Učící faktor - určuje, jak moc se budou v každé epoše váhy upravovat (desetinné číslo větší než 0)
- Počet epoch - určuje, kolikrát se bude opakovat proces učení (celé číslo větší než 0)

Proces učení můžeme rozšířit o několik dalších parametrů:

- Dávka - udává počet vzorů po kterých dojde k úpravě vah, tento parametr ovlivňuje jak často se budou upravovat váhy (celé číslo větší než 0)
- Momentum - zavádí do procesu učení "paměť změn", parametr určuje jak moc bude minulá změna vah ovlivňovat současnou změnu vah (desetinné číslo větší nebo rovno 0, 0 = žádná "paměť")

3.2 Fáze učení

Nejdříve nastavíme všechny váhy a biasy na náhodné malé hodnoty. Následně začíná učení. Fáze učení pomocí algoritmu CD se pro každý vektor, který chceme síť naučit, skládá z těchto kroků:

1. Stavy viditelných neuronů v_i nastavíme na stejné hodnoty, jako má vstupní vektor. Každý viditelný neuron si svůj stav zapamatuje jako v'_i.
2. Aktualizujeme stavy všech skrytých neuronů následujícím způsobem. Nejdříve si podle rovnice 2 spočítáme aktivační energii daného skrytého neuronu j a následně aktualizujeme stav neuronu h_j s pravděpodobností $\sigma(h_j)$ na 1 a s pravděpodobností $1-\sigma(h_j)$ na 0. Pravděpodobnost $\sigma(x)$ se vypočte podle rovnice 3. Každý skrytý neuron si svůj stav zapamatuje jako h'_j.

$$h_j = b_j + \sum_{i=1}^{N} w_{ij} v_i \tag{2}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

3. Pro každý spoj w_{ij} vypočteme hodnotu $Positive(w_{ij}) = v_i * h_j$.
4. Nyní analogickým způsobem aktualizujeme stavy všech viditelných neuronů. Nejdříve si podle rovnice 4 spočítáme aktivační energii daného viditelného neuronu i a následně aktualizujeme stav neuronu v_i s pravděpodobností $\sigma(v_i)$ na 1 a s pravděpodobností $1-\sigma(v_i)$ na 0. Výsledné stavy viditelných neuronů se můžou lišit od těch původních. Každý viditelný neuron si svůj stav zapamatuje jako v''_i.

$$v_i = a_i + \sum_{j=1}^{M} w_{ij} h_j \tag{4}$$

5. Opět aktualizujeme stavy všech skrytých neuronů stejným způsobem jako v bodě 2. Každý skrytý neuron si svůj stav zapamatuje jako h''_j.
6. Pro každý spoj w_{ij} vypočteme hodnotu $Negative(w_{ij}) = v_i * h_j$.

7. Aktualizujeme váhy všech spojů podle rovnice 5 a všech biasů podle rovnice 6, kde α představuje učící faktor.

$$w_{ij} = w_{ij} + \alpha(Positive(w_{ij}) - Negative(w_{ij})) \tag{5}$$

$$a_i = a_i + \alpha(v'_i - v''_i) \quad b_j = b_j + \alpha(h'_j - h''_j) \tag{6}$$

Výše uvedený postup zopakujeme pro všechny vektory trénovací množiny. Tímto je dokončena první epocha učení. Celý postup potom opakujeme tak dlouho, jak nám určuje zadaný počet epoch. Aktualizaci vah nemusíme provádět ihned, ale můžeme ji provádět až po několika vektorech trénovací množiny. Počet vektorů, po kterých se provádí aktualizace vah, potom určuje parametr dávka. Proces učení můžeme také rozšířit o momentum. Pro toto rozšíření je nutné si u každé váhy pamatovat přírůstek. Aktualizace vah se poté provede podle rovnice 7, kde δ označuje minulý přírůstek a M označuje parametr momentum.

$$w_{ij} = w_{ij} + \alpha(Positive(w_{ij}) - Negative(w_{ij})) + M\delta_{ij} \tag{7}$$

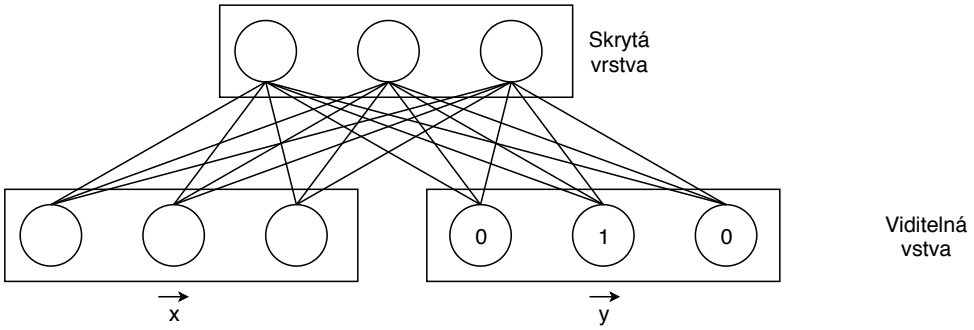
3.3 Fáze vybavování

Tento proces je narozdíl od procesu učení jednorázový a ne iterační. Funguje na podobném principu jako učení. Fáze vybavení probíhá následujícím způsobem:

1. Stavy viditelných neuronů v_i nastavíme na stejné hodnoty jako má vstupní vektor.
2. Aktualizujeme stavy všech skrytých neuronů stejně jako v bodě 2 ve fázi učení.
3. Aktualizujeme stavy všech viditelných neuronů stejně jako v bodě 4 ve fázi učení.
4. Jednotlivé stavy viditelných neuronů v_i odpovídají našemu vybavenému vektoru.

3.4 Rozšíření RBM pro klasifikaci

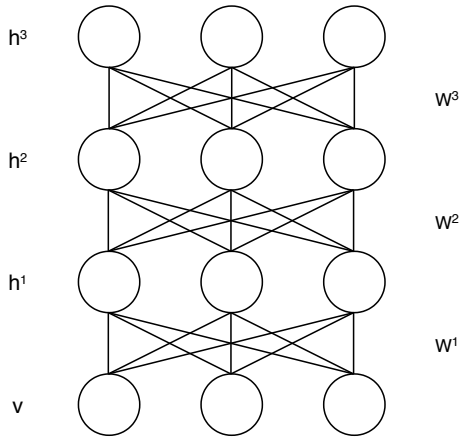
RBM jsou nejčastěji při klasifikaci používány pouze k modelování vstupů, ale můžeme je použít také pro samotnou klasifikaci, tedy pro modelování vstupů a patřičných klasifikačních tříd [17]. Vytvoříme trénovací množinu $D_{train} = \{(x_i, y_i)\}$, která se skládá z jednotlivých vstupních vektorů x_i a binárních vektorů $y_i \in \{1, ..., C\}$, kde C představuje počet tříd. Tento binární vektor y_i má jedničku na pozici třídy do které spadá daný vstupní vektor x_i a na ostatních pozicích má nuly. RBM poté naučíme trénovací množinu D_{train} a při fázi vybavování následně můžeme využít vektor y_i pro klasifikaci. Výsledný model RBM je zobrazen na obrázku 2.



Obrázek 2: RBM pro klasifikaci

4 Deep Boltzmann Machine

Deep Boltzmann Machine (DBM) [23, 24, 22] je vícevrstvá neuronová síť. První vrstva se nazývá viditelná a ostatní vrstvy se nazývají skryté. V případě binárního modelu mohou všechny neurony nabývat hodnot $\{0, 1\}$. Každý neuron je propojen se všemi neurony sousední vyšší a nižší vrstvy. Neurony v rámci stejné vrstvy spolu propojeny nejsou. Každý neuron má navíc ještě svůj bias. Všechny váhy sítě jsou symetrické, tedy $w_{ij} = w_{ji}$. Struktura této sítě s jednou viditelnou a třemi skrytými vrstvami je zobrazena na obrázku 3.



Obrázek 3: Ukázka struktury DBM

Každá konfigurace neuronů má svou energii, která je dána rovnicí 8, kde v je vektor stavů viditelných neuronů, h^i je vektor stavů skrytých neuronů ve skryté vrstvě i, L počet vrstev skrytých neuronů a W^i váhy spojů ve skryté vrstvě i.

$$E(v, h^1, h^2, ..., h^L) = -v^\top W^1 h^1 - h^{1\top} W^2 h^2 - ... - h^{L-1\top} W^L h^L \tag{8}$$

4.1 Parametry sítě

Algoritmus učení DBM se skládá ze dvou hlavních fází - předtrénování a trénování. Vyžaduje několik vstupních parametrů:

- Počet viditelných neuronů
- Počet vrstev skrytých neuronů a počty skrytých neuronů v jednotlivých vrstvách
- Učící faktor pro fázi předtrénování a trénování
- Počet epoch pro fázi předtrénování a trénování
- Koeficient snižování učícího faktoru ve fázi trénování
- Počet kroků Gibbsova vzorkování ve fázi trénování

- Počet Mean-field kroků ve fázi trénování
- Počet náhodně generovaných vzorkovacích částic ve fázi trénování

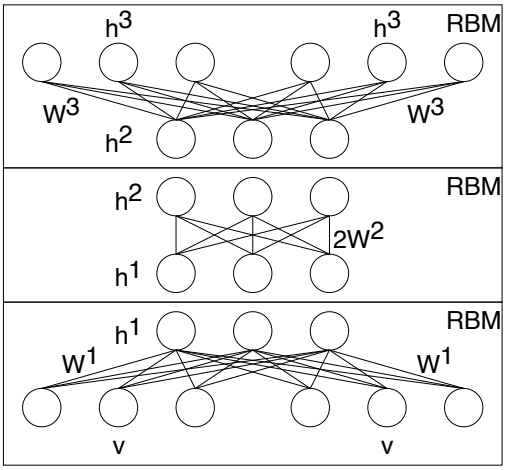
Proces učení můžeme opět rozšířit o parametry momentum a dávka a to jak ve fázi předtrénování, tak i ve fázi trénování.

4.2 Fáze učení

Učení se skládá se dvou fází - předtrénování a trénování.

4.2.1 Předtrénování

Předtrénování probíhá postupně po jednotlivých vrstvách. Jedná se o učení jednotlivých RBM pomocí algoritmu CD s drobnými změnami. Při učení nejnižší vrstvy DBM je nutné zdvojit vstupy (viditelné neurony), což vede k tomu, že při průchodu zdola nahoru vstupují do skrytých neuronů dvojnásobné hodnoty. Stejně tak při učení nejvyšší vrstvy DBM je nutné zdvojit skryté neurony nejvyšší vrstvy, což vede k tomu, že při průchodu shora dolů vstupují do skrytých neuronů předposlední vrstvy dvojnásobné hodnoty. V případě učení mezilehlých vrstev DBM je nutné zdvojit váhy, aby se dvojnásobné hodnoty vyskytovaly v obou směrech. Princip zdvojování pro DBM se třemi skrytými vrstvami zobrazuje obrázek 4. Po dokončení předtrénování všechna zdvojení odstraníme.



Obrázek 4: Princip zdvojování při předtréninku DBM

Jednotlivé kroky pro DBM se třemi skrytými vrstvami pro fázi předtrénování tedy jsou:

1. Naučit první vrstvu DBM pomocí algoritmu CD pro RBM se zdvojenými viditelnými neurony na trénovací množině A.
2. Pomocí algoritmu vybavení RBM vytvořit z původní trénovací množiny A novou trénovací množinu B pro další vrstvu DBM.

3. Naučit druhou vrstvu DBM pomocí algoritmu CD pro RBM se zdvojenými vahami na novou trénovací množinu B.

4. Pomocí algoritmu vybavení RBM vytvořit z trénovací množiny B novou trénovací množinu C pro další vrstvu DBM.

5. Naučit třetí vrstvu DBM pomocí algoritmu CD pro RBM se zdvojenými skrytými neurony na novou trénovací množinu C.

6. Výsledkem předtrénování jsou váhy $\theta_0 = \{W^1, W^2, W^3\}$.

4.2.2 Trénování

Tato fáze algoritmu učení vyžaduje dvě oddělené množiny vah. Nejdříve tedy vytvoříme pomocné váhy $\theta_0^{rec} = \{R^1, R^2, R^3\}$, které vzniknou jako kopie vah $\theta_0 = \{W^1, W^2, W^3\}$. Každá epocha se skládá ze tří po sobě jdoucích částí:

1. Variational interference

2. Stochastic approximation

3. Aktualizace vah

Variational interference

Následující kroky opakujeme pro všechny prvky v_n trénovací množiny:

1. Provedeme průchod sítí zdola nahoru pomocí rovnic 9, 10 a 11, kde D představuje počet viditelných neuronů a F_i počet skrytých neuronů ve skryté vrstvě i. Při výpočtu všech vrstev kromě vrstvy nejvyšší používáme ve výpočtu $2R$. Výsledkem tohoto průchodu je vektor parametrů $\nu = \{\nu^1, \nu^2, \nu^3\}$.

$$\nu_j^1 = \sigma(\sum_{i=1}^{D} 2R_{ij}^1 v_i) \tag{9}$$

$$\nu_k^2 = \sigma(\sum_{j=1}^{F_1} 2R_{jk}^2 \nu_j^1) \tag{10}$$

$$\nu_m^3 = \sigma(\sum_{k=1}^{F_2} R_{km}^3 \nu_k^2) \tag{11}$$

2. Provedeme přiřazení $\mu = \nu$. Provedeme K iterací mean-field aktualizací (parametr algoritmu) podle rovnic 12, 13 a 14. Výsledný vektor μ si zapamatujeme jako μ_n, kde n představuje index prvku v trénovací množině.

$$\mu_j^1 \leftarrow \sigma(\sum_{i=1}^{D} W_{ij}^1 v_i + \sum_{k=1}^{F_2} W_{jk}^2 \mu_k^2) \tag{12}$$

$$\mu_k^2 \leftarrow \sigma(\sum_{j=1}^{F_1} W_{jk}^2 \mu_j^1 + \sum_{m=1}^{F_3} W_{km}^3 \mu_m^3) \tag{13}$$

$$\mu_m^3 \leftarrow \sigma(\sum_{k=1}^{F_2} W_{km}^3 \mu_k^2) \tag{14}$$

3. Provedeme úpravu vah θ^{rec} pomocí rovnice 15, kde α_t je učící faktor v čase t. Tento gradient lze spočítat pomocí algoritmu Backpropagation, který zde nebudu popisovat.

$$\theta_{t+1}^{rec} = \theta_t^{rec} + \alpha_t \frac{\partial(-\sum_i \mu_i \log \nu_i - \sum_i (1-\mu_i) \log (1-\nu_i))}{\partial \theta^{rec}} \tag{15}$$

Stochastic approximation

Náhodně vygenerujeme M vzorkovacích částic $(\bar{v}_{t,m}, \bar{h}_{t,m})$, kde M je vstupní parametr algoritmu, m označuje index vygenerované vzorkovací částice a $\bar{h} = \{\bar{h}^1, \bar{h}^2, \bar{h}^3\}$. Pro všechny vygenerované vzorkovací částice provedeme Gibbsovo vzorkování s počtem kroků daných vstupním parametrem. Výsledkem vzorkování pro každou částici bude vektor $(\bar{v}_{t+1,m}, \bar{h}_{t+1,m})$, kde m označuje index částice. Gibbsovo vzorkování se provádí pomocí stejných rovnic jako mean-field s tím rozdílem, že vzorkujeme navíc také vrstvu viditelných neuronů.

Aktualizace vah

Upravíme váhy θ sítě DBM podle rovnic 16, 17 a 18. Po úpravě vah snížíme učící koeficient α tak, že ho vynásobíme koeficientem snižování učícího faktoru.

$$W_{t+1}^1 = W_t^1 + \alpha_t(\frac{1}{N}\sum_{n=1}^{N} v_n(\mu n^1)^\top - \frac{1}{M}\sum_{m=1}^{M} \bar{v}_{t+1,m}(\bar{h}_{t+1,m}^1)^\top) \tag{16}$$

$$W_{t+1}^2 = W_t^2 + \alpha_t(\frac{1}{N}\sum_{n=1}^{N} \mu_n^1(\mu n^2)^\top - \frac{1}{M}\sum_{m=1}^{M} \bar{h}_{t+1,m}^1(\bar{h}_{t+1,m}^2)^\top) \tag{17}$$

$$W_{t+1}^3 = W_t^3 + \alpha_t(\frac{1}{N}\sum_{n=1}^{N} \mu_n^2(\mu n^3)^\top - \frac{1}{M}\sum_{m=1}^{M} \bar{h}_{t+1,m}^2(\bar{h}_{t+1,m}^3)^\top) \tag{18}$$

4.3 Fáze vybavení a použití DBM pro klasifikaci

Fáze vybavení DBM probíhá úplně stejně jako v případě RBM, jen je nutné postupně projít přes všechny skryté vrstvy tam a zpět.

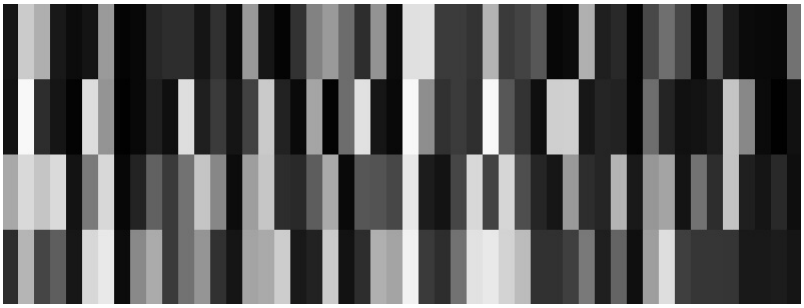
Takto vytvořený DBM lze následně pomocí standardního algoritmu Backpropagation doladit tak, aby uměl provádět klasifikaci. Nicméně já jsem ve své práci použil pro klasifikaci stejný přístup rozšíření jako byl popsán v případě RBM.

5 Monitorování učení neuronové sítě

Algoritmy učení neuronových sítí vyžadují několik vstupních parametrů, které významně ovlivňují výslednou kvalitu naučení sítě. Vizualizace, popsané v této kapitole, nám mohou přiblížit chování sítě během procesu učení a také nám mohou pomoci vhodně zvolit vstupní parametry. Autoři v publikaci [26] tyto vizualizace aplikovali na RBM. V této práci jsem tyto vizualizace aplikoval na RBM i DBM.

5.1 Pravděpodobnost aktivace skrytých neuronů

Každý skrytý neuron má pro daný vstupní vzor určitou pravděpodobnost aktivace. Tato pravděpodobnost je v intervalu $< 0, 1 >$. Vizualizace těchto pravděpodobností nám umožní vidět, jak často jsou dané skryté neurony využívány. Tyto pravděpodobnosti můžeme efektivně vizualizovat pomocí obrázku ve stupních šedi, kde každý řádek obsahuje pravděpodobnosti pro konkrétní vstupní vzor a každý sloupec obsahuje pravděpodobnosti pro konkrétní skrytý neuron napříč jednotlivými vstupními vzory. Bílá barva reprezentuje hodnotu pravděpodobnosti 1 a černá barva hodnotu pravděpodobnosti 0. Ukázku této vizualizace zobrazuje obrázek 5.



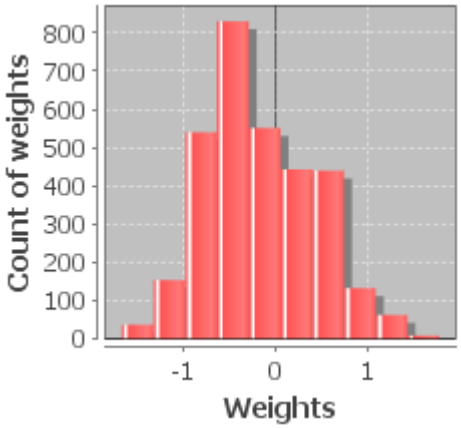
Obrázek 5: Vizualizace pravděpodobnosti aktivace skrytých neuronů

Před započetím učení by tento obrázek měl být téměř jednolitě šedý, což znamená že se pravděpodobnosti aktivace pohybují kolem hodnoty $0,5$ a počáteční hodnoty vah byly správně vygenerovány. Vhodné je si tyto obrázky pravděpodobností z jednotlivých epoch spustit za sebou jako animaci. V animaci můžeme vidět postupný vývoj těchto pravděpodobností napříč jednotlivými epochami. V případě, že máme nastavený učící faktor na příliš vysokou hodnotu, tak v animaci uvidíme blikání jednotlivých pravděpodobností trvající po několik epoch.

5.2 Histogramy vah

Dalším druhem vizualizace jsou histogramy vah. Konkrétně se jedná o histogram vah, biasů viditelných neuronů a biasů skrytých neuronů. Užitečný je nejen histogram samotných hodnot, ale také histogram přírůstků těchto hodnot za danou epochu. Všechny histogramy by měly mít Gaussovo rozdělení. V případě, že během učení dochází k příliš malým změnám vah, tak můžeme

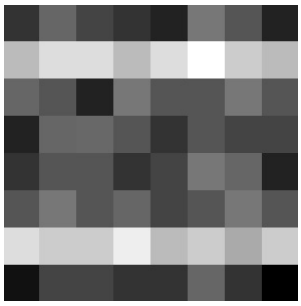
pravděpodobně zvýšit hodnotu učícího faktoru. Ukázku vizualizace histogramu vah zobrazuje obrázek 6.



Obrázek 6: Ukázka histogramu vah

5.3 Filtry

Pro každý skrytý neuron můžeme zobrazit jeho filtr. Filtry jsou nejčastěji vizualizovány stejným způsobem jako vstupní data, tedy v případě 2D obrázku jako 2D obrázky. Každý filtr je potom obrázek ve stupních šedi stejné dimenze jako vstupní data. Filtr nám zobrazuje, jak viditelné neurony ovlivňují aktivaci konkrétního skrytého neuronu. Jedná se o vizuální reprezentaci vah vedoucích od viditelných neuronů do konkrétního skrytého neuronu. Bílá barva reprezentuje nejvyšší váhu a černá barva nejnižší váhu. Na základě těchto filtrů můžeme zjistit, zda daný skrytý neuron reaguje pouze na několik málo vstupních neuronů a detekuje tak nějaký lokální rys, případně zda reaguje na velké množství viditelných neuronů napříč celým filtrem. Ukázku filtru detekující konkrétní rys zobrazuje obrázek 7.

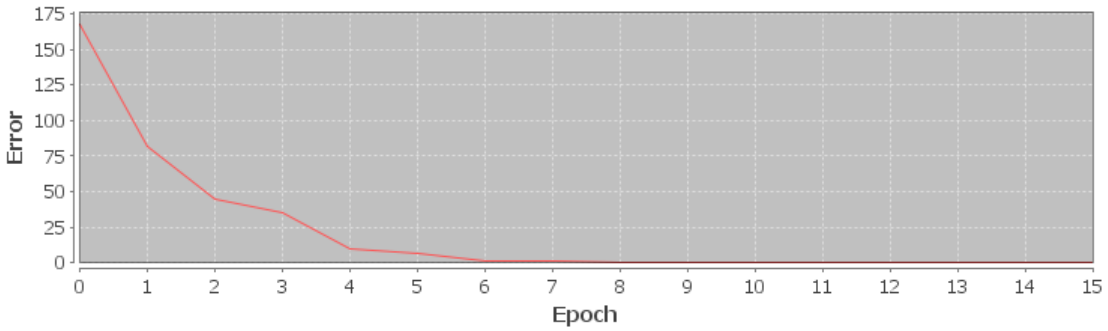


Obrázek 7: Ukázka filtru

5.4 Chyba

Dalším užitečným druhem vizualizace je graf vývoje chyby napříč jednotlivými epochami. V tomto grafu jsou na horizontální ose epochy a na vertikální ose je počet chyb. Počet chyb by měl po-

stupně v průběhu učení klesat. Jako chybu můžeme brát špatně rozpoznaný vstupní vzor, případně každý špatně rekonstruovaný pixel obrázku. Tento graf nám může napovědět, zda máme zadaný dostatečný počet epoch, případně zda nemáme epoch zbytečně příliš hodně. Ukázku grafu zobrazuje obrázek 8.



Obrázek 8: Ukázka grafu vývoje chyby

6 Paralelizace

Učení neuronových sítí vyžaduje velký výpočetní výkon. Jednou z možností, jak tohoto výkonu dosáhnout je využití paralelizace učení neuronové sítě [10]. Na paralelismus můžeme nahlížet ze dvou pohledů, a to jako na paralelismus v rámci jednoho stroje a paralelismus na více strojích.

Paralelismus v rámci jednoho stroje je v dnešní počítačové architektuře všudypřítomný. Setkáme se s ním například v případě vícejádrových procesorů a vláken. Z důvodu potřeby velkého výpočetního výkonu ale jeden stroj často není schopen dokončit zadaný úkol v přijatelném čase. Abychom zvýšili dostupný výpočetní výkon, jsme nuceni výpočet distribuovat mezi několik strojů vzájemně propojených pomocí počítačové sítě.

Rozdílné síťové technologie nám poskytují rozdílný výkon a rychlost komunikace mezi jednotlivými uzly je v případě distribuovaného výpočtu velmi důležitá. Pro tento účel byly navržené speciální komunikační sítě pro HPC (High-performance computing) jako je například InfinitiBand, která má velmi vysokou propustnost a velmi nízké zpoždění. Nicméně i přesto síťová komunikace zůstává obecně pomalejší než komunikace v rámci jednoho stroje.

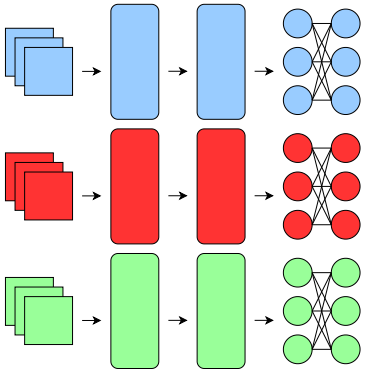
V rámci této diplomové práce jsem se zabýval paralelizací na více strojích (uzlech). V následujících podkapitolách budou popsány existující strategie pro paralelizaci neuronových sítí, paralelizaci jejich učení a protokoly pro komunikaci po síti.

6.1 Strategie pro paralelizaci neuronové sítě

Existují tři hlavní strategie pomocí kterých můžeme neuronovou síť paralelizovat. Jedná se o paralelizaci na úrovni vstupních souborů (datový paralelismus), síťové struktury (modelový paralelismus) a vrtev (zřetězené zpracování). Jejich zkombinováním potom dostaneme čtvrtou strategii nazývanou hybridní paralelismus.

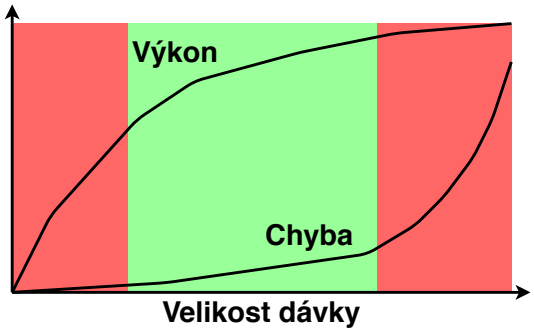
6.1.1 Datový paralelismus

V případě této strategie rozdělíme vstupní soubor na několik dávek, kde každá dávka bude obsahovat N vstupních vzorů. Neuronovou síť budeme následně učit po těchto jednotlivých dávkách, kdy každou dávku vždy přidělíme nějakému výpočetnímu zdroji (jádru nebo uzlu). Každý uzel si poté uchovává svou celou kopii neuronové sítě na které probíhá učení na přidělené dávce. Váhy těchto sítí je potřeba po každé dávce synchronizovat. Rychlost synchronizace těchto vah napříč výpočetními uzly je velmi důležitá. Schéma této strategie je zobrazeno na obrázku 9. Barvy představují jednotlivé výpočetní uzly, čtverce vstupy a obdélníky jednotlivé fáze algoritmu učení, případně vrstvy neuronové sítě.



Obrázek 9: Datový paralelismus

Klíčové je nastavení správné velikosti dávky, protože velikost dávky ovlivňuje kvalitu naučení sítě i efektivitu paralelizace. Pokud bude velikost dávky příliš malá, tak bude často docházet k synchronizaci vah a celkový výkon bude nízký. Pokud bude naopak velikost dávky příliš velká, tak bude velmi málo docházet k synchronizaci vah a v důsledku toho bude kvalita naučení sítě nízká. Vzájemný vztah mezi výkonem, chybou sítě (kvalitou naučení) a velikostí dávky zobrazuje obrázek 10.

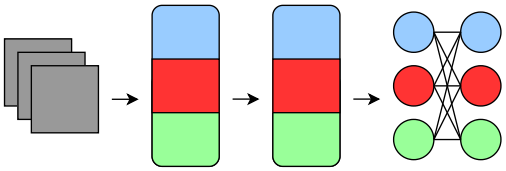


Obrázek 10: Vliv velikosti dávky na chybu a výkon

6.1.2 Modelový paralelismus

Modelový paralelismus funguje tak, že celou neuronovou síť rozdělí mezi jednotlivé uzly. Dojde k rozdělení neuronové sítě na několik skupin neuronů. Každý vstupní vzor je zaslán na všechny uzly a každý uzel poté provádí výpočty týkající se pouze přidělené skupiny neuronů. Rozdílné části neuronové sítě jsou tak počítány na rozdílných uzlech. Toto může vést k menším nárokům na paměť, protože není třeba mít celou síť uloženou na jednom uzlu, a k větší potřebě komunikace jednotlivých uzlů z důvodu závislostí mezi jednotlivými skupinami neuronů při výpočtu. Pro omezení potřeby příliš velké komunikace v případě velkých závislostí mezi jednotlivými skupinami neuronů můžeme například zavést redundantní výpočty. V tomto případě pak dochází k tomu, že se určité stejné výpočty provádí paralelně na více uzlech namísto toho, aby se daný výpočet provedl pouze na jednom uzlu a výsledek se ostatním uzlům, které jej potřebují, zaslal.

Tím, že se paralelizace neprovádí na úrovni vstupních dat jako tomu bylo u datového paralelismu, nedochází ke zhoršení kvality naučení sítě. Schéma této strategie je zobrazeno na obrázku 11.



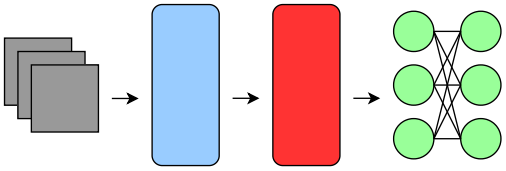
Obrázek 11: Modelový paralelismus

6.1.3 Zřetězené zpracování

Zřetězené zpracování leží na pomezí datového a modelového paralelismu. Můžeme na něj nahlížet jako na datový paralelismus, kdy vstupní vzory jsou napříč sítí zpracovávány paralelně, ale také jako na modelový paralelismus, kdy jednotlivé komponenty řetězu jsou dány strukturou neuronové sítě. V prvním případě rozdělíme algoritmus učení na jednotlivé fáze a každou fázi přidělíme nějakému výpočetnímu uzlu. Každý vstupní vzor je potom postupně zřetězeně zpracováván ve všech výpočetních uzlech. V druhém případě rozdělíme síť po jednotlivých vrstvách a každou vrstvu přidělíme nějakému výpočetnímu uzlu. To s sebou přináší několik výhod:

- Není třeba mít uloženou celou síť na jednom uzlu.
- Pevný počet bodů pro komunikaci mezi uzly (na hranicích vrstev).
- Vždy známý zdrojový a cílový uzel.

Nevýhodou zřetězeného zpracování může být, že vstupní data musí přicházet určitou rychlostí a výpočty jednotlivých komponent musí trvat přibližně stejně dlouho, aby bylo možné plně využít všechny výpočetní uzly. Schéma této strategie je zobrazeno na obrázku 12.



Obrázek 12: Zřetězené zpracování

6.1.4 Hybridní paralelismus

Pomocí různých kombinací tří výše uvedených strategií můžeme vytvořit strategii čtvrtou, nazývanou hybridní paralelismus. Můžeme například zkombinovat všechny tři strategie následujícím způsobem. Učení bude probíhat na několika kopiích neuronové sítě paralelně, kde každá kopie

bude učena na jiné dávce a mezi kopiemi bude docházet k synchronizaci vah (datový paralelismus). Učení v rámci jedné kopie neuronové sítě bude distribuováno mezi uzly po jednotlivých vrstvách (zřetězené zpracování) a v rámci jedné vrstvy bude dále distribuováno na základě jednotlivých skupin neuronů (modelový paralelismus). Datový paralelismus a zřetězené zpracování může být například řešeno na úrovni jednotlivých uzlů a modelový paralelismus pak na úrovni jednotlivých jader procesoru v rámci jednoho uzlu.

6.1.5 Volba strategie

Každá z výše uvedených strategií má své výhody a nevýhody. V neuronových sítích RBM a DBM je poměrně velká závislost mezi jednotlivými neurony. Použití modelového paralelismu mi tudíž nepřišlo vhodné z důvodu potřeby velké komunikace mezi jednotlivými uzly. Navíc experimenty jsem neplánoval se sítěmi s až tak velkými počty neuronů, aby se jednotlivé neurony daly efektivně rozdělit mezi jednotlivé uzly.

Použití zřetězeného zpracování by bylo v případě sítě RBM také komplikované, protože se síť skládá pouze z jedné vrstvy a jednotlivé kroky algoritmu jsou poměrně úzce provázány a tudíž by si bylo nutné mezi uzly předávat velké množství mezivýsledků. V případě sítě DBM by se zřetězené zpracování aplikovat dalo, ale chtěl jsem zachovat jednotný přístup paralelizace u obou neuronových sítí.

V rámci této práce jsem se rozhodl pro strategii datového paralelismu, protože mi přišel nejvhodnější. Pro tuto strategii je charakteristický vzájemný vztah mezi výkonem, chybou sítě a zvolenou velikostí dávky. Experimenty v závěru této práce odhalí k jak velkému zvýšení výkonu a chybovosti neuronových sítí RBM a DBM v důsledku datového paralelismu dojde.

6.2 Strategie pro synchronizaci vah

V předchozích podkapitolách jsme žádným způsobem neřešili distribuci vah mezi jednotlivými uzly. Rozdělení výpočtu mezi více uzlů v případě datového paralelismu s sebou ale přináší potřebu, aby všechny uzly mezi sebou sdílely určitým způsobem váhy a udržovala se tak konzistence modelu neuronové sítě napříč uzly. Jednotlivé uzly si tak mezi sebou musí vyměňovat provedenou deltu vah. Při výměně delty vah je důležité co nejvíce minimalizovat velikost zprávy. Toho můžeme docílit pomocí komprese zprávy a také tak, že budeme ve zprávě posílat pouze nutné informace. Výměnu změny vah můžeme realizovat pomocí několika strategií:

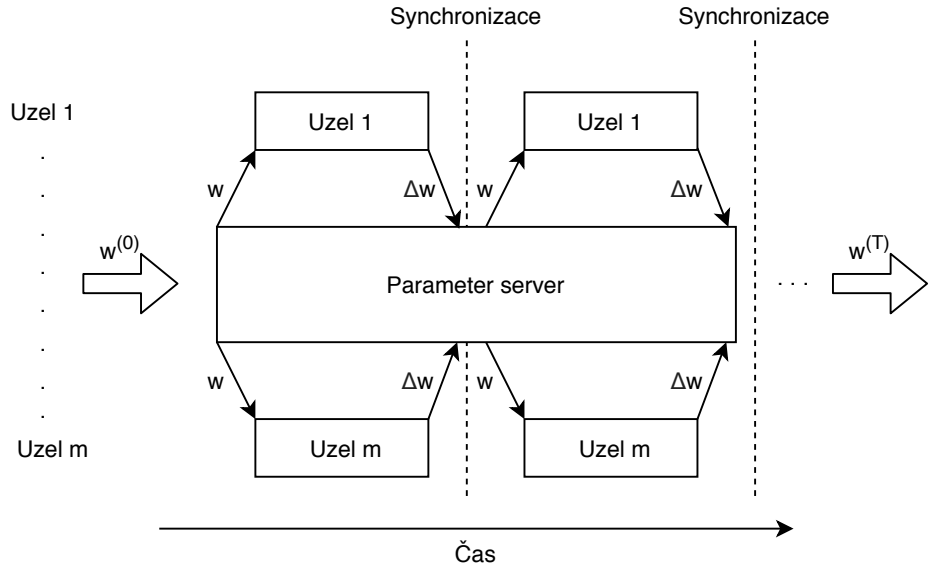
- Synchronní centralizovaný přístup
- Asynchronní centralizovaný přístup
- Synchronní decentralizovaný přístup

6.2.1 Synchronní centralizovaný přístup

V případě této strategie budeme potřebovat Parameter server (PS). Tento speciální uzel bude spravovat globální verzi vah, provádět aktualizace vah na základě přijatých delt vah a také bude tyto aktualizované váhy zasílat ostatním uzlům. Tato architektura je pouze abstraktní koncept. PS nemusí být nutně reprezentovaný jedním fyzickým uzlem. Můžeme vytvořit více těchto serverů a každý z nich poté může spravovat pouze určitou část vah, což je užitečné v případě použití kombinace datového a modelového paralelismu. Další užitečnou vlastností centralizovaného přístupu je, že umožňuje jednoduše dynamicky přidávat a odebírat uzly během procesu učení. Tím, že máme nejaktuálnější společnou verzi vah na jednom místě, můžeme snadno zvýšit odolnost proti chybám. Můžeme periodicky ukládat verzi vah společně s informací o počtu již proběhlých epoch na disk a v případě pádu programu takto uložená data využít k tomu, že nemusíme začít celý proces učení neuronové sítě od úplného začátku.

Tato strategie funguje následujícím způsobem. PS vždy všem ostatním uzlům zašle aktuální globální verzi vah. Každý uzel následně provádí učení neuronové sítě na mu přidělené dávce dat a ukládá si deltu úpravy vah. Po dokončení učení na přidělené dávce uzel zašle PS spočítanou deltu vah. PS následně začlení přijatou deltu do své globální verze vah tak, že tuto deltu přičte ke svým spravovaným vahám. Takto postupně PS začlení delty vah od jednotlivých uzlů do své globální verze. Po začlenění delt vah od všech uzlů PS rozešle aktualizovanou verzi vah všem uzlům a celý proces se opakuje.

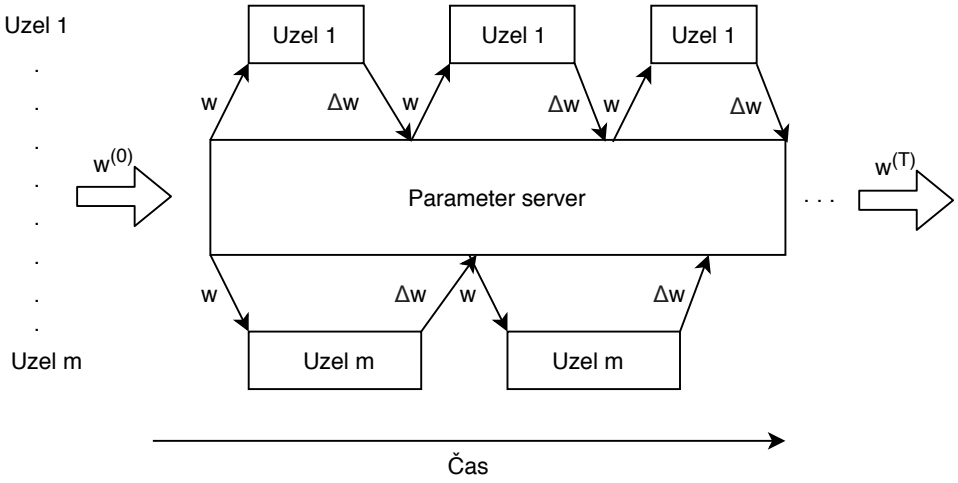
Tato strategie se nazývá synchronní, protože synchronizace vah probíhá najednou u všech uzlů, a centralizovaná, protože synchronizace vah probíhá na jednom místě a to na PS. Schéma této strategie je zobrazeno na obrázku 13.



Obrázek 13: Centralizovaný synchronní přístup

6.2.2 Asynchronní centralizovaný přístup

Tato strategie opět využívá centralizace za pomocí Parameter serveru (PS). Funguje podobně jako Synchronní centralizovaný přístup. Rozdíl je v tom, že PS nečeká po začlenění delty vah od uzlu m do svých spravovaných globálních vah na delty vah od ostatních uzlů, ale ihned uzlu m zašle aktualizovanou verzi vah. Nedochází tak k tomu, že by jednotlivé uzly na sebe musely po dokončení výpočtu navzájem čekat. Tato strategie se nazývá asynchronní, protože nedochází k synchronizaci vah napříč všemi uzly v jeden konkrétní okamžik. Schéma této strategie je zobrazeno na obrázku 14.



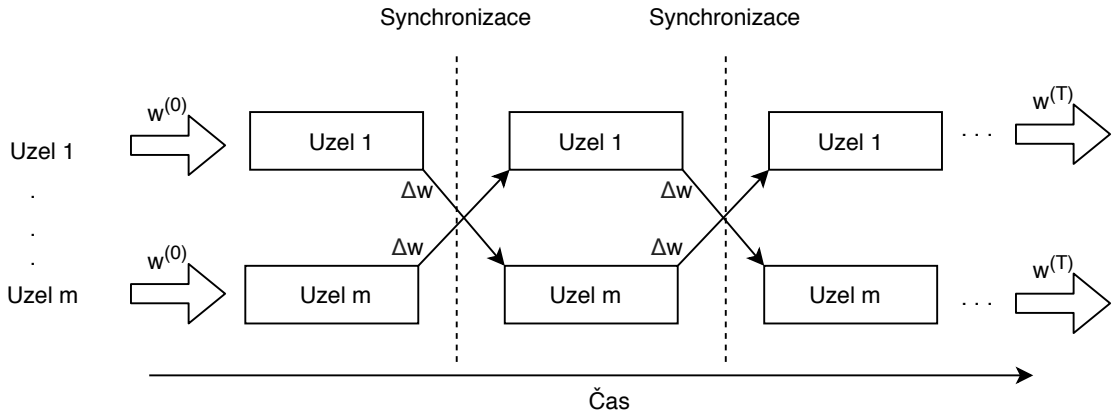
Obrázek 14: Centralizovaný asynchronní přístup

6.2.3 Synchronní decentralizovaný přístup

V případě této strategie již nepoužíváme žádný Parameter server, který by spravoval globální verzi vah. Jednotlivé uzly tak musí po každé zpracované dávce navzájem provést synchronizaci vah. Každý uzel musí zaslat každému uzlu svou vypočtenou deltu vah a všechny uzly pak musí tyto delty vah přičíst ke své verzi vah.

Tato strategie funguje následujícím způsobem. Každý uzel provádí učení neuronové sítě na mu přidělené dávce dat a ukládá si deltu úpravy vah. Po dokončení učení na přidělené dávce uzel zašle všem ostatním uzlům svou vypočtenou deltu vah. Následně uzel vyčkává na delty vah od ostatních uzlů a tyto přijaté delty přičítá ke své verzi vah. Po začlenění vah od všech uzlů uzel začne provádět učení na další dávce dat a celý proces se opakuje.

Tato strategie se nazývá synchronní, protože synchronizace vah probíhá najednou u všech uzlů, a decentralizovaná, protože synchronizaci vah nemá na starost nějaký centrální prvek. Schéma této strategie je zobrazeno na obrázku 15.



Obrázek 15: Decentralizovaný synchronní přístup

6.2.4 Volba strategie

V rámci této práce jsem se rozhodl pro asynchronní centralizovaný přístup. Použití decentralizovaného přístupu mi nepřišlo vhodné z důvodu poměrně komplikované synchronizace vah. Navíc jsem chtěl mít v rámci procesu učení nějaký centrální prvek, který bude jednotlivým uzlům přidělovat jednotlivé dávky dat pro učení. V budoucnu by se dal navíc centrální prvek, díky jeho globálnímu pohledu na aktuální stav procesu učení, využít například pro monitorování průběhu učení. Asynchronní přístup jsem zvolil z důvodu rychlosti, protože jednotlivé uzly na sebe nebudou muset po dokončení výpočtu na zadané dávce čekat.

6.3 Komunikační protokoly

Jelikož distribuovaný výpočet probíhá na několika různých strojích a tyto stroje mezi sebou musí provádět synchronizaci vah, tak tyto stroje spolu musí komunikovat prostřednictvím počítačové sítě. Rychlost a spolehlivost této komunikace je velmi důležitá. Příliš pomalá komunikace může způsobit to, že jednotlivé uzly stráví většinu času zasíláním, přijímáním a čekáním na zprávy. V současnosti patří mezi nejpoužívanější protokoly pro komunikaci po síti protokoly TCP a UDP.

6.3.1 UDP

UDP (User Datagram Protocol) [13] je protokol transportní vrstvy modelu TCP/IP. Protokol UDP pro identifikaci odesílatele a příjemce na konkrétním stroji používá číslo portu. Pakety přenášené tímto protokolem se nazývají datagramy. Tento protokol neposkytuje žádný mechanismus potvrzování pro úspěšně přijaté zprávy. V případě, že je detekována chyba v přijatém paketu, tak je paket zahozen a odesílatel o tom není nijak informován. Každý odeslaný datagram je nezávislý a tedy neexistuje žádný vztah mezi odeslanými datagramy a to i přesto, že pochází od stejného odesílatele a jsou určeny pro stejného příjemce. Také se na začátku a konci přenosu nenavazuje a neukončuje spojení jako je tomu u protokolu TCP. Z tohoto důvodu může

každý datagram putovat po jiné cestě. Výhodou tohoto protokolu je jeho jednoduchost, která způsobuje minimální režii při komunikaci a tudíž vyšší rychlost.

6.3.2 TCP

TCP (Transmission Control Protocol) [13] je stejně jako UDP protokol transportní vrstvy modelu TCP/IP. Také pro identifikaci odesílatele a příjemce na konkrétním stroji používá číslo portu. Přenos dat v případě TCP funguje tak, že nejdříve se naváže spojení mezi odesílatelem a příjemcem, poté probíhá přenos samotné zprávy a nakonec se spojení ukončí. Všechny segmenty dané zprávy jsou tak díky navázanému spojení přenášeny po jedné virtuální cestě. Tento protokol poskytuje mechanismus potvrzování pro úspěšně přijaté zprávy. V případě, že daný segment nedorazí nebo dorazí poškozen, tak je zaslán znovu. Pokud jednotlivé segmenty dorazí příjemci ve špatném pořadí, tak se TCP postará o jejich seřazení. TCP má oproti UDP spoustu výhod, ale jedná se o komplikovanější protokol s vyšší režií a tudíž nižší rychlostí.

6.3.3 Volba protokolu

V případě distribuovaného výpočtu je u komunikace důležitá rychlost a spolehlivost. Protokol UDP nám poskytuje vyšší rychlost na úkor nižší spolehlivosti a TCP zase vyšší spolehlivost na úkor nižší rychlosti. Rozhodl jsem se pro protokol UDP, protože komunikace bude probíhat v relativně malé počítačové síti, kde datům nebude hrozit příliš mnoho komplikací, a spolehlivost lze v případě chyby při přenosu vyřešit případným opakovaným zasláním zprávy. Dalším důvodem je, že jsem se distribuovaný výpočet rozhodl stavět na již existujícím prototypu knihovny, které vzniklo v rámci diplomové práce Ing. Vojtěcha Pustówky [20]. Tento prototyp knihovny právě využívá protokol UDP.

7 Implementace

Tato kapitola se zaměřuje na popis implementační části diplomové práce. Jsou zde popsány použité technologie a knihovny, datasety a program Modeler neuronových sítí včetně podrobného popisu rozšíření vytvořeného v rámci této práce.

7.1 Použité technologie a knihovny

Při implementaci programů jsem využil programovací jazyk Java, nástroje Maven a GIT a knihovny Aeron a JFreeChart. Všechny tyto technologie a knihovny jsou popsány v následujících podkapitolách.

7.1.1 Java

Programovací jazyk Java jsem použil z důvodu, že je v něm naimplementován program Modeler neuronových sítí i prototyp knihovny pro paralelizaci neuronových sítí. Java [3, 4] je jeden z nejpoužívanější programovacích jazyků na světě. Jedná se o objektově orientovaný jazyk, který vyvinula firma Sun Microsystems. První verze jazyka Java byla představena v roce 1995. Samotný jazyk vychází z jazyka C++. Výhodou Javy je její přenositelnost a platformní nezávislost. Při kompilaci zdrojového kódu se vytváří ByteCode, který je potom spouštěn na virtuálním stroji Javy. Program tedy stačí napsat pouze jednou a následně lze spouštět všude tam, kde je nainstalován příslušný virtuální stroj. Současným oficiálním vlastníkem implementace Java SE je od roku 2010 společnost Oracle.

7.1.2 Maven

Maven [6] je nástroj pro správu, řízení a automatizaci buildů aplikací. Používá se především v kombinaci s programovacím jazykem Java. Nástroj Maven mimo jiné umožňuje v souboru pom.xml u každého projektu nadefinovat jeho závislosti na externích knihovnách. Maven pak automaticky potřebné knihovny vyhledá v definovaných uložištích a následně je nainstaluje.

7.1.3 GIT

GIT [12] je distribuovaný systém správy verzí vyvinutý v roce 2005. Jedná se o snadno použitelný, rychlý a efektivní systém. Jeho úkolem je zaznamenávat změny souboru nebo sady souborů v průběhu času a umožnit tak uživateli kdykoliv obnovit konkrétní verzi těchto sledovaných souborů. GIT umožňuje také jednoduchou spolupráci více vývojářů na jednom projektu, čehož bylo využito v rámci této práce, protože v tomto akademickém roce rozšiřovalo program Modeler neuronových sítí více studentů.

7.1.4 JFreeChart

JFreeChart [5] je knihovna pro programovací jazyk Java, která umožňuje jednoduše vizualizovat data formou grafů. Obsahuje Swing i JavaFX komponenty a umožňuje z grafu vytvářet obrázky formátu PNG, JPEG, PDF a další. Podporuje spoustu různých grafů jako například sloupcový graf, čárový graf, koláčový graf, bodový graf a další.

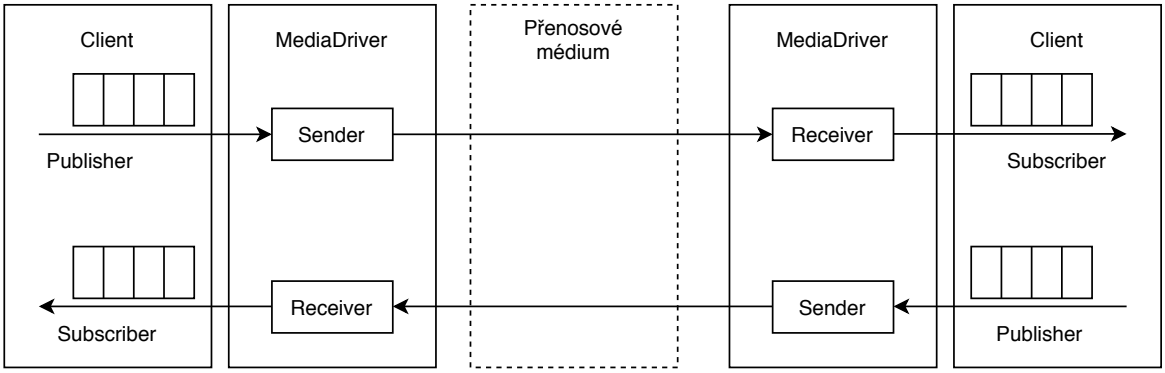
Práci s touto knihovnou lze demonstrovat na příkladu vytvoření sloupcového grafu. Nejdříve je potřeba vytvořit dataset, který následně naplníme daty, které chceme zobrazit v grafu. V případě sloupcového grafu můžeme využít třídu $DefaultCategoryDataset$. Z vytvořeného a naplněného datasetu je dále potřeba vytvořit samotný graf třídy $JFreeChart$. Pro jeho vytvoření použijeme třídu $ChartFactory$. Výsledný graf nakonec můžeme zobrazit pomocí komponenty $ChartPanel$. Celý tento postup je zobrazen v ukázce kódu 1.

```
1 DefaultCategoryDataset dataset = new DefaultCategoryDataset();
2 dataset.addValue(10, "", "Kategorie 1");
3 dataset.addValue(20, "", "Kategorie 2");
4 JFreeChart barChart = ChartFactory.createBarChart("Název grafu", "Popis osy X", "Popis osy Y", dataset,
      PlotOrientation.VERTICAL, false, false, false);
5 ChartPanel chart = new ChartPanel(barChart);
```

Výpis 1: Ukázka použití knihovny JFreeChart

7.1.5 Aeron

Aeron [1] je knihovna pro komunikaci po síti. Je dostupná pro Javu, C++ a .NET. Umí pracovat v režimu unicast i multicast a klade důraz na vysokou propustnost a nízké zpoždění. Aeron umí pracovat nad různými typy přenosových médií jako je například UDP, TCP, InfinityBand a další. MediaDriver potom implementuje podporu pro jednotlivé protokoly. Architektura této knihovny je zobrazena na obrázku 16.



Obrázek 16: Knihovna Aeron

Práce s knihovnou Aeron je relativně jednoduchá. Každý účastník komunikace (odesílatel a příjemce) je identifikován pomocí identifikátorů Channel a Stream. Channel identifikuje sa-

motného účastníka komunikace a zadává se pomocí textového řetězce. V tomto řetězci uvádíme IP adresu, port a použitý protokol. Ukázka takového Channel řetězce je zobrazena v kódu 2 na řádku 1. Stream můžeme využít pro identifikaci v rámci jednoho účastníka a zadává se pomocí celého čísla.

Při práci s knihovnou Aeron je nutné nejdříve vytvořit instanci třídy *MediaDriver*. Poté můžeme vytvořit samotného klienta pomocí třídy *Aeron*. Příjem zprávy probíhá pomocí třídy *Subscription*, která naslouchá na uvedeném identifikátoru Channel a Stream. Samotné zpracování přijatých dat provedeme v callbacku třídy *FragmentHandler*. Během odesílání větších zpráv automaticky dochází k fragmentaci zprávy na jednotlivé datové rámce. Na straně příjemce je tyto fragmentované zprávy nutné znovu poskládat pomocí třídy *FragmentAssembler*. Ukázka jednoduchého příjemce je zobrazena v kódu 2. Odeslání zprávy na zadaný Channel a Stream provedeme jednoduše pomocí třídy *Publication*. Ukázka jednoduchého odesílatele je zobrazena v kódu 3.

```
1  String channel = "aeron:udp?endpoint=192.168.0.1:40456";
2  int stream = 1;
3  final MediaDriver driver = MediaDriver.launchEmbedded();
4  Aeron aeron = Aeron.connect(new Aeron.Context().aeronDirectoryName(driver.aeronDirectoryName()));
5  final Subscription subscription = aeron.addSubscription(channel, stream);
6  final FragmentHandler fragmentHandler = (buffer, offset, length, header) -> {
7      final byte[] data = new byte[length];
8      buffer.getBytes(offset, data);
9      System.out.println(new String(data));
10 };
11 FragmentHandler fragmentAssembler = new FragmentAssembler(fragmentHandler);
12 while(true) {
13     int fragmentsRead = subscription.poll(fragmentAssembler, 10);
14 }
```

Výpis 2: Příjemce v knihovně Aeron

```
1  String channel = "aeron:udp?endpoint=192.168.0.1:40456";
2  int stream = 1;
3  String message = "Hello World!";
4  final MediaDriver driver = MediaDriver.launchEmbedded();
5  Aeron aeron = Aeron.connect(new Aeron.Context().aeronDirectoryName(driver.aeronDirectoryName()));
6  final Publication publication = aeron.addPublication(channel, stream);
7  final UnsafeBuffer buffer = new UnsafeBuffer(BufferUtil.allocateDirectAligned(256, 64));
8  buffer.putBytes(0, message.getBytes());
9  long result = publication.offer(buffer, 0, message.getBytes().length);
```

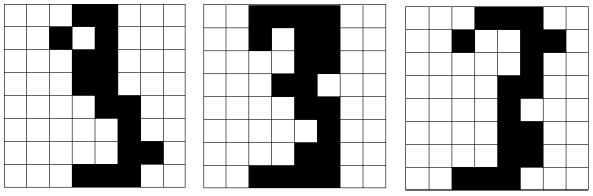
Výpis 3: Odesilatel v knihovně Aeron

7.2 Použité datasety

Do programu Modeler neuronových sítí jsem naimplementoval možnost pracovat s datasety OPT digits, MNIST, CalTech 101 Silhouettes a OCR letters. Program umí také pracovat s datasety ve formátu CSV. Zmíněné datasety jsou popsány v následujících podkapitolách.

7.2.1 OPT digits

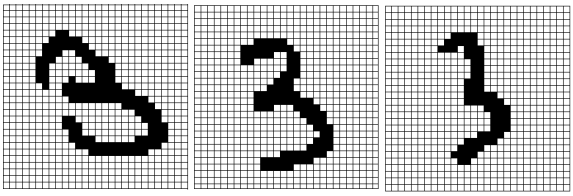
Dataset OPT digits [9] je pro klasifikaci číslic 0 až 9. Úkolem je tedy každý vzor správně zařadit do jedné z 10 tříd. Obsahuje trénovací množinu o velikosti 3823 vzorů a testovací množinu o velikosti 1797 vzorů. Byl vytvořen na základě ručně psaných číslic od 43 lidí. Každý vzor má velikost 8x8 a každý element je celé číslo z intervalu $< 0, 16 >$. Pro potřeby binárních modelů neuronových sítí RBM a DBM v rámci této práce jsem hodnotám větším než 8 přiřadil 1 a hodnotám menším nebo rovným 8 přiřadil 0. Ukázka několika různých číslic 3 tohoto datasetu je zobrazena na obrázku 17.



Obrázek 17: Ukázka číslic 3 datasetu OPT

7.2.2 MNIST

MNIST [7] je další dataset pro klasifikaci číslic 0 až 9. Obsahuje trénovací množinu o velikosti 60000 vzorů a testovací množinu o velikosti 10000 vzorů. Každý vzor má velikost 28x28 a každý element je celé číslo z intervalu $< 0, 255 >$. Hodnotám větším než 140 jsem přiřadil 1 a hodnotám menším nebo rovným 140 přiřadil 0. Ukázka několika různých číslic 3 tohoto datasetu je zobrazena na obrázku 18.

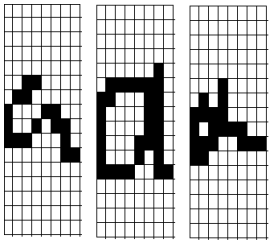


Obrázek 18: Ukázka číslic 3 datasetu MNIST

7.2.3 OCR letters

Dataset OCR letters [8] je pro klasifikaci malých písmen abecedy a až z. Úkolem je každý vzor správně zařadit do jedné z 26 tříd. Dataset obsahuje 52152 vzorů. Z těchto vzorů jsem vytvořil

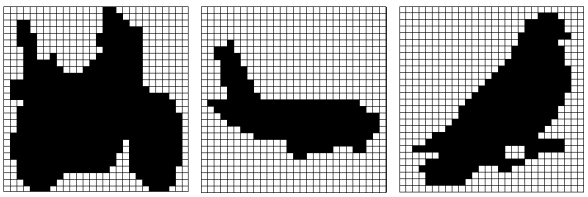
trénovací množinu o velikosti 41734 vzorů a testovací množinu o velikosti 10418 vzorů. Testovací množina tak obsahuje přibližně 20% vzorů od každého znaku. Každý vzor má velikost 16x8 a každý element nabývá jedné z binárních hodnot 0 nebo 1. Ukázka několika různých znaků a tohoto datasetu je zobrazena na obrázku 19.



Obrázek 19: Ukázka znaků a datasetu OCR

7.2.4 CalTech 101 Silhouettes

Dataset CalTech 101 Silhouettes je pro klasifikaci siluet objektů. Úkolem je každý vzor správně zařadit do jedné ze 101 tříd. Jedná se vždy o černou siluetu objektu na bílém pozadí. Jedná se například o siluety letadel, mravenců, bobrů, deštníků a spoustu dalších. Autor datasetu uvádí, že některé třídy jako třeba fotbalový míč a pizza jsou od sebe na základě siluety nerozeznatelné, ale přesto je v datasetu ponechal. Dataset je k dispozici ve dvou verzích, a to ve velikosti vzorů 28x28 a 16x16. Každý element v obou verzích nabývá jedné z binárních hodnot 0 nebo 1. Trénovací množina ve verzi 16x16 obsahuje 4082 vzorů a ve verzi 28x28 4100 vzorů. Testovací množina ve verzi 16x16 obsahuje 2302 vzorů a ve verzi 28x28 2307 vzorů. Dataset je k dispozici ve formě Matlab souborů. Pro potřeby této práce jsem z těchto souborů pomocí Matlabu vytvořil textové soubory. Ukázka vzorů reprezentujících motocykl, letadlo a holuba (v tomto pořadí z leva) je zobrazena na obrázku 20.



Obrázek 20: Ukázka siluet datasetu CalTech 101 Silhouettes

7.3 Program Modeler neuronových sítí

Aplikace Modeler neuronových sítí je simulátor neuronových sítí s podporou vizualizace vnitřní struktury, učení a vybavení dat, který je především určen pro podporu výuky předmětu Neuronové sítě. Program umožňuje pomocí grafického uživatelského rozhraní jednoduše vytvořit konkrétní neuronovou síť, kterou je možné následně naučit na zvoleném datasetu. Uživatel tak nemusí mít znalosti z oblasti programování, aby mohl s neuronovými sítěmi pracovat. Program

je napsán v programovacím jazyce Java, tudíž pro spuštění programu je nutné mít Javu nainstalovanou v počítači. Samotný program není třeba instalovat.

V dosavadní verzi programu bylo několik typů neuronových sítí. Důležitou část programu tvoří balíček *neuronNetModeler*, který obsahuje společné části programu a definuje několik rozhraní, které poté musí implementovat konkrétní neuronové sítě. Dále se v programu nachází několik dalších balíčků, které implementují různé neuronové sítě a různé další funkcionality. Mezi důležitá rozhraní hlavního balíčku *neuronNetModeler* patří:

- $AxonInterface$ - definuje metody pro axon (vstup neuronové sítě)
- $ConnectionInterface$ - definuje metody pro spoj mezi neurony
- $NetObjectExtendInterface$ - definuje metody pro položku v rozbalovacím stromě v hlavním okně programu
- $NeuronNetInterface$ - definuje metody pro samotnou neuronovou síť
- $NetConstructorInterface$ - definuje metody pro třídu pomocí které se vytváří daná neuronová síť
- $NeuronInterface$ - definuje metody pro neuron

7.4 Implementace rozšíření programu

Rozšíření programu můžeme rozdělit do tří oblastí. Jedná se o samotnou implementaci neuronových sítí RBM a DBM, implementaci nástrojů pro vizualizaci sítí a práci s datasety a implementaci komunikace s programem pro paralelní část. Toto rozšíření je strukturováno do 4 balíčků a to:

- $neuronNetModeler.RestrictedBoltzmannMachine$
- $neuronNetModeler.DeepBoltzmannMachine$
- $neuronNetModeler.MonitoringAndTools$
- $neuronNetModeler.ParallelLearningConfig$

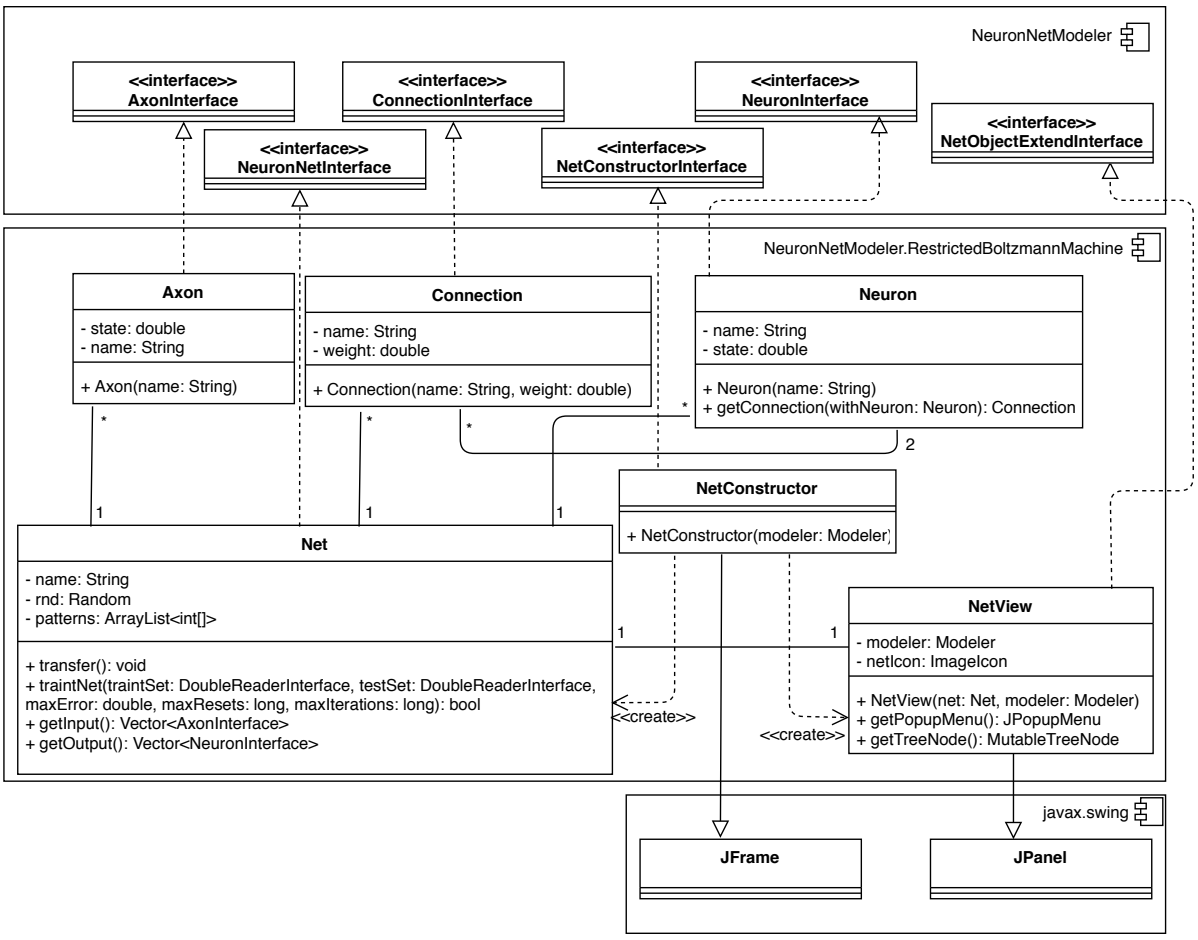
7.4.1 Implementace neuronových sítí

Implementace každé neuronové sítě je ve vlastním balíčku. Síť RBM se nachází v balíčku neuronNetModeler.RestrictedBoltzmannMachine a síť DBM se nachází v balíčku neuronNetModeler.DeepBoltzmannMachine. Oba dva balíčky mají stejnou strukturu. První verzi implementace balíčku pro RBM zobrazuje třídní diagram na obrázku 21. Celá neuronová síť je vytvořená pomocí zobrazených tříd, kde každá třída implementuje jí určené rozhraní.

Nejdůležitější třídou ve které se nachází algoritmus učení a vybavení je třída Net. Pro učení neuronové sítě slouží veřejná metoda $trainNet$ a pro vybavení metoda $transfer$. Před zavoláním metody $transfer$ je nutné nejdříve neuronové síti předložit patřičný vstup. Ke vstupním axonům získáme přístup pomocí metody $getInput$. Po vykonání metody $transfer$ se výsledek nachází v jednotlivých neuronech. Přístup k těmto neuronům získáme pomocí metody $getOutput$. Všechny tyto metody vychází z rozhraní $NeuronNetInterface$.

Dalšími důležitými třídami jsou třídy $NetConstructor$ a $NetView$. $NetConstructor$ zajišťuje nastavení parametrů pro neuronovou síť a vytvoření objektu třídy Net. Třída $NetView$ slouží pro zobrazení sítě na úvodní obrazovce programu. Také je v ní definované vyskakovací menu, které slouží pro další práci s neuronovou sítí v rámci programu.

Tato objektová reprezentace celé neuronové sítě se ukázala již během rané fáze testování z výkonnostních důvodů jako nevhodná.



Obrázek 21: UML třídní diagram první verze implementace neuronové sítě RBM

Algoritmus učení pracoval s objekty tříd $Neuron$ a $Connection$ a testování ukázalo, že tento objektový přístup je velmi pomalý a nepraktický. Z tohoto důvodu vznikla nová verze implementace, která již nevyužívá pro uložení samotné neuronové sítě objekty. Místo objektů se v současné

implementaci používají vícerozměrné pole. Pro přístup ke konkrétnímu prvku neuronové sítě se používají indexy dané váhy, případně neuronu, a pro přístup ke konkrétní proměnné v rámci tohoto prvku (v případě neuronu například bias, stav, případně další proměnné potřebné v rámci algoritmu učení) se jako indexy používají předdefinované konstanty.

Z původního třídního diagramu tak byla úplně vynechána třída *Connection*. Třídy *Axon* a *Neuron* byly zachovány. Tyto třídy se nadále používají při komunikaci se zbylou již existující částí programu, ale nepoužívají se v rámci algoritmů učení a vybavení. Třída *Axon* slouží pouze pro případné nastavení vstupu a třída *Neuron* pro případné poskytnutí výstupu. Třídy *Axon*, *Neuron* i *Net* nemají implementaci některých metod, protože tyto metody nedávají v případě neuronových sítí RBM a DBM a v uvedeném přístupu použití smysl.

Důsledkem této změny implementace došlo u neuronové sítě RBM k výraznému zrychlení procesu učení. Porovnání rychlostí je uvedeno v tabulce 1. Toto testování bylo provedeno na notebooku na neuronové síti o počtu 784 viditelných neuronů, 200 skrytých neuronů, 30 vstupních vzorech a 100 epochách. Z tabulky 1 jde vidět, že čas učení po změně implementace je přibližně poloviční. Uvedené zrychlení je ovšem vykoupeno menší přehledností a horší orientací v kódu. Z výše uvedených důvodů je neuronová síť DBM naimplementována také pomocí vícerozměrných polí.

Typ implementace	**Čas v sekundách**
Objekty	85
Vícerozměrné pole	38

Tabulka 1: Porovnání rychlosti procesu učení různých implementací sítě RBM

7.4.2 Implementace nástrojů

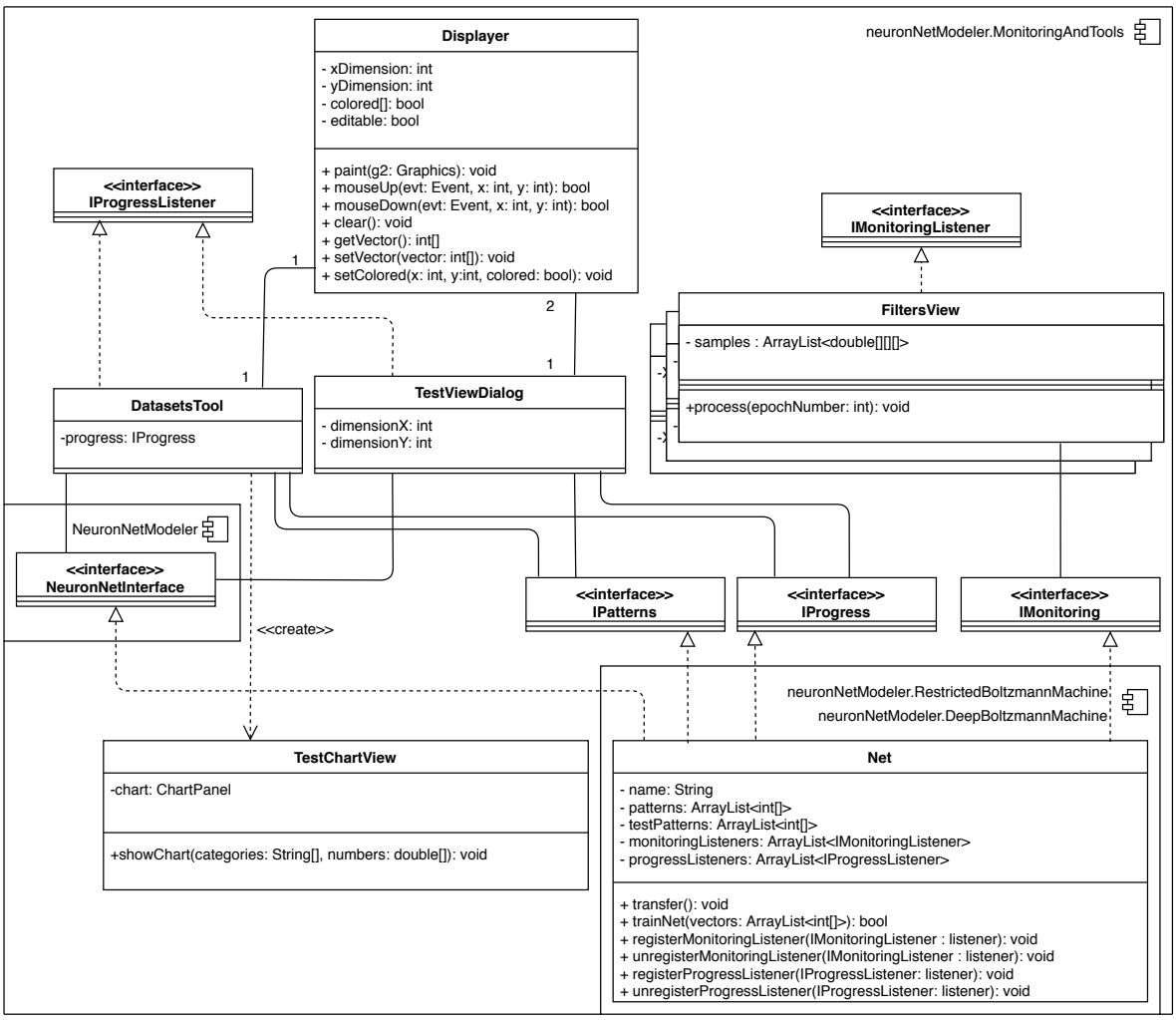
Implementace nástrojů se nachází v balíčku neuronNetModeler.MonitoringAndTools. V tomto balíčku se také nachází všechna potřebná rozhraní, která musí neuronová síť implementovat, aby mohla tyto nástroje využívat. Třídy implementující nástroje popsané v této podkapitole a související rozhraní jsou znázorněny pomocí třídního diagramu na obrázku 22. Nástroje lze rozdělit do dvou skupin, a to na nástroje pro práci s datasety a nástroje pro monitorování průběhu učení.

První skupinou nástrojů jsou nástroje pro práci s datasety. Pro práci s datasety slouží třídy *DatasetsTool*, *TestChartView*, *TestViewDialog* a *Displayer*. Třída *DatasetsTool* zajišťuje práci se všemi datasety uvedenými v kapitole 7.2 Použité datasety. Navíc ještě umí načíst libovolný uživatelsky definovaný dataset ve formátu CSV. Třída má spoustu různých funkcí. Umí uvedené datasety načíst a umožní uživateli přiřadit libovolné vzory z těchto datasetů do trénovací a testovací množiny. Dále umí zobrazit uživatelem zvolený vzor k čemuž využívá třídu *Displayer*. Poslední funkcí této třídy je zobrazení úspěšnosti rozpoznání testovací množiny formou tabulky, případně použitím třídy *TestChartView* formou grafu.

Jak již bylo zmíněno výše, tak třída *DatasetsTool* využívá třídy $TestChartView$ a *Displayer*. Třída $TestChartView$ umí zobrazit výsledky úspěšnosti rozpoznání testovací množiny formou sloupcového grafu. K vytvoření tohoto grafu používá knihovnu JFreeChart. Třída *Displayer* slouží pro vizualizaci jednotlivých vzorů. Po nastavení rozměrů mřížky a předložení vzoru ve formě pole umí tato třída tento vzor v mřížce zobrazit. Další funkcionalitou této třídy je, že umožňuje uživateli do této mřížky kreslit vlastní vzory s využitím myši. Této vlastnosti využívá třída $TestViewDialog$.

Třída $TestViewDialog$ umožňuje uživateli vytvořit dataset o zvolené velikosti mřížky s využitím myši přímo v programu a následně na tento dataset neuronovou síť naučit. Využívá k tomu právě výše popsanou třídu *Displayer*. Také uživateli umožňuje zobrazit odpověď neuronové sítě na předložený vzor.

Jak třída $TestViewDialog$, tak třída *DatasetsTool* umí zobrazit pomocí ukazatele průběhu aktuální stav právě probíhající operace (učení nebo vybavení). Danou informaci o aktuálním stavu získává ze třídy Net pomocí návrhového vzoru EventListener.



Obrázek 22: UML třídní diagram implementace nástrojů

Druhou skupinou nástrojů jsou nástroje pro monitorování průběhu učení. Jedná se o vizualizační nástroje popsané v kapitole 5 Monitorování učení neuronové sítě. Každou z těchto vizualizací zajišťuje jedna třída. Data pro vytváření těchto vizualizací v průběhu učení neuronové sítě tyto třídy získávají ze třídy *Net* pomocí návrhového vzoru EventListener. Je tak nutné před samotným spuštěním procesu učení požadované vizualizace zapnout a zaregistrovat k neuronové síti. Vizualizační nástroje pak průběžně zobrazují aktuální stav dané vizualizace. Některé třídy využívají k zobrazení vizualizace třídu *Canvas* a některé knihovnu JFreeChart.

7.4.3 Implementace komunikace s programem pro paralelizaci

Další potřebnou funkcionalitou programu je komunikace s programem pro paralelizaci neuronových sítí. Tato komunikace probíhá pomocí konfiguračního a datového souboru. Vytváření těchto souborů zajišťuje třída *ParallelLearning*, která se nachází v balíčku neuronNetModeler.MonitoringAndTools. Tato třída umožňuje uživateli nastavit velikost dávky pro paralelní učení, počet epoch v rámci jedné dávky, IP adresu a port uzlu typu Master a IP adresy a porty uzlů typu Slave.

Samotný konfigurační soubor je soubor typu XML. Jsou v něm obsaženy všechny informace týkající se paralelizace, typu a struktury neuronové sítě a parametry potřebné pro algoritmus učení. Tento soubor se vytváří serializací třídy $ParallelLearningConfig$, která v sobě má informace týkající se paralelizace a taky obsahuje třídu $NetParameters$. Tato třída v sobě nese informaci o typu neuronové sítě a jejich parametrech. Obě dvě třídy se nachází v balíčku neuronNetModeler.ParallelLearningConfig.

Instanci třídy $NetParameters$ vytváří přímo konkrétní neuronová síť. *ParallelLearning* si od dané sítě tuto instanci vyžádá pomocí rozhraní $IParameters$. Naopak instanci třídy $ParallelLearningConfig$ vytváří *ParallelLearning* a navíc zajišťuje serializaci této třídy do souboru XML.

Dalším souborem jehož vytváření zajišťuje třída *ParallelLearning* je soubor nesoucí dataset pro učení. Jedná se o objekt obsahující jednotlivé vzory pro učení serializovaný do binární podoby. Tento objekt si *ParallelLearning* vyžádá přímo od konkrétní neuronové sítě pomocí rozhraní $IPatterns$. Tento přístup zajišťuje to, že na straně programu pro paralelizaci je již na vstupu jednotný formát datasetu. Na straně programu pro paralelizaci se nemusí řešit různé formáty datasetů a tuto problematiku tak řeší jen třída *DatasetsTool* v programu Modeler neuronových sítí.

Program pro paralelizaci provede na základě těchto dvou vstupních souborů učení a výstupem tohoto programu je soubor obsahující naučené váhy. Jedná se opět o serializovaný binární objekt. Načítání tohoto souboru v programu Modeler neuronových sítí zajišťuje třída $NetView$ dané neuronové sítě a do sítě tyto naučené váhy vkládá pomocí rozhraní $NetExport$. Toto rozhraní vyžaduje po neuronové síti implementaci metod pro import i export vah. Díky tomu je možné binární soubor obsahující váhy vytvořit také ze sítě v modeleru.

7.5 Implementace paralelizace

Pro paralelizaci neuronových sítí RBM a DBM jsem zvolil datový paralelismus v kombinaci s asynchronním centralizovaným přístupem a komunikačním protokolem UDP. Popis implementace programu pro paralelizaci lze rozdělit do dvou oblastí, a to na implementaci samotných uzlů typu Master a Slave a na implementaci různých módů spouštění programu, zejména pak implementaci automatického určení role uzlu.

7.5.1 Implementace uzlů Master a Slave

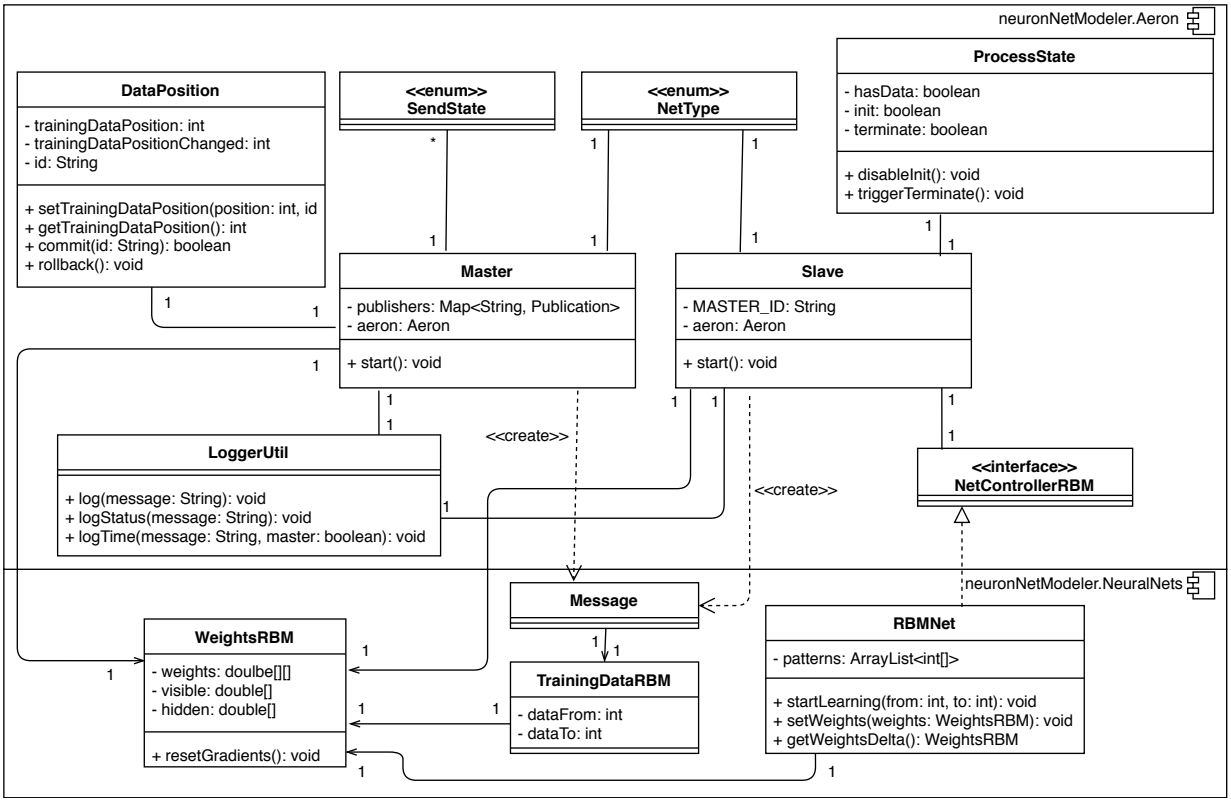
Při implementaci uzlů Master a Slave jsem použil knihovnu Aeron. Jako základ implementace programu byla použita knihovna vytvořená v rámci diplomové práce Ing. Vojtěcha Pustówky. Tuto knihovnu jsem upravil pro potřeby neuronových sítí RBM a DBM. Dále jsem se zabýval efektivitou zasílání zpráv a původně dva samostatné programy Master a Slave spojil do jednoho. Program se skládá z následujících balíčků:

- $neuronNetModeler.Aeron$
- $neuronNetModeler.NeuralNets$
- $neuronNetModeler.ParallelLearningConfig$

Balíček neuronNetModeler.ParallelLearningConfig obsahuje stejné třídy jako v programu Modeler neuronových sítí a slouží pro komunikaci mezi těmito dvěma programy. Způsob této komunikace byl popsán v podkapitole 7.4.3 Implementace komunikace s programem pro paralelizaci. V balíčku neuronNetModeler.Aeron jsou třídy jejichž úkolem je samotná komunikace jednotlivých uzlů a v balíčku neuronNetModeler.NeuralNets se nachází třídy představující neuronové sítě a třídy reprezentující zasílanou zprávu pro tyto sítě. Přehled důležitých tříd je zobrazen formou UML třídního diagramu na obrázku 23.

Třída *Master* je hlavní řídící prvek celého programu. V instanci *Master* je obsažen seznam všech aktivních instancí *Slave* společně s jejich aktuálním stavem. Tato třída neustále ve smyčce kontroluje jednotlivé instance typu *Slave* a v případě potřeby jim zasílá novou verzi vah a identifikátory prvků trénovací množiny pro učení. Udržuje globální verzi vah, kterou rozesílá všem instancím typu *Slave*, a delty vah přijaté od těchto instancí do této globální verze integruje. Po ukončení učení z této globální verze vah vytvoří výstupní soubor obsahující naučené váhy.

Třída *Master* přiřazuje jednotlivé dávky dat pro učení konkrétním instancím *Slave* pomocí třídy *DataPosition*. Tato třída si udržuje informaci o naposledy úspěšně odeslané dávce dat. Master tak může jednoduše posunout ukazatel odeslaných dat na další pozici a v případě úspěšného odeslání tuto pozici potvrdit pomocí metody commit. V případě, že odeslání selže, tak může tuto novou pozici ukazatele vrátit zpět pomocí metody rollback. Tímto mechanismem je zajištěno, že v případě selhání odeslání zprávy se v rámci další zprávy odešle znovu stejná dávka dat.



Obrázek 23: UML třídní diagram implementace programu pro paralelizaci

Další důležitou třídou je třída *Slave*. *Slave* přijímá od instance $Master$ zprávy obsahující vždy novou verzi vah a identifikátory vzorů pro učení pro aktuální dávku. Tuto novou verzi vah následně předá samotné neuronové síti. Poté spustí učení neuronové sítě na přidělené množině trénovacích dat. Po dokončení učení si od neuronové sítě vyžádá deltu vah a tuto deltu zašle zpět instanci $Master$. Následně vyčkává na další pokyn od instance $Master$. V případě, že další zpráva v požadovaném časovém intervalu od instance $Master$ nedorazí, tak *Slave* zašle instanci $Master$ novou zprávu, protože původní zpráva se pravděpodobně z nějakého neznámého důvodu ztratila.

Třída $RBMNet$ (případně $DBMNet$) na základě požadavků od třídy *Slave* provádí samotný algoritmus učení. Při procesu učení si pamatuje vzniklé delty vah, které pak na vyžádání předá třídě *Slave*.

Třídy $Master$ a *Slave* používají pro logování třídu *LoggerUtil*. Pomocí této třídy lze logovat různé zprávy do textového souboru nebo na konzoli. Pro vzájemné předávání informací třídy $Master$ a *Slave* používají třídu $Message$. Tato třída pak v sobě obsahuje další třídy, které v sobě mají samotné váhy neuronové sítě a identifikátory trénovacích dat pro přidělenou dávku.

Samotný proces odeslání zprávy se skládá z několika kroků. Nejdříve je vytvořena a naplněna konkrétními daty instance třídy $Message$. Tato třída je následně serializována a zkomprimována tak, aby měla co nejmenší velikost. Takto zkomprimovaná zpráva je nakonec odeslána požado-

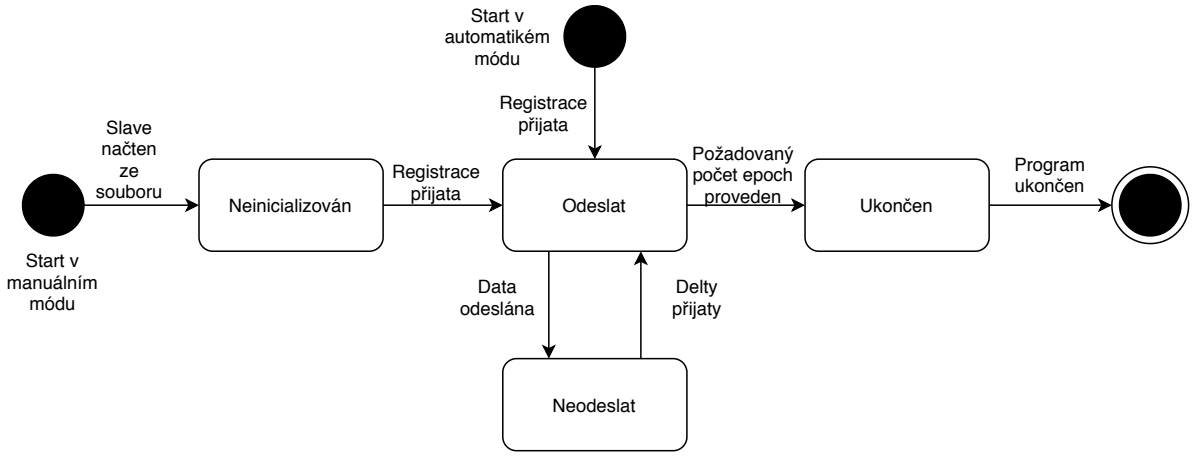
vanému adresátovi. V původní verzi knihovny od Ing. Vojtěcha Pustówky se v rámci zprávy posílaly jak váhy, tak trénovací množina pro danou dávku. Posílání přímo trénovací množiny ale zbytečně zvyšovalo velikost zprávy. Zvolil jsem proto přístup, kdy si každý uzel načte celou trénovací množinu a *Master* následně v rámci zprávy zasílá pouze interval trénovacích vzorů, které se má daný *Slave* v této dávce učit. Pro účely tohoto přístupu vznikla již výše popsaná třída *DataPosition*.

V původní verzi knihovny se serializace zprávy prováděla do textového formátu JSON. Tento přístup se také ukázal vzhledem k velikosti zpráv jako neefektivní. Daleko menší velikosti zprávy lze dosáhnout pouhou serializací samotného objektu do pole bytů. Porovnání velikosti zpráv demonstruje následující experiment jehož výsledky jsou zobrazeny v tabulce 2. V rámci tohoto experimentu jsem náhodně vygeneroval pole o velikosti 1000 desetinných čísel datového typu *double*. Toto pole jsem následně serializoval s využitím třídy *Gson* do textu formátu JSON a s využitím tříd *ByteArrayOutputStream* a *ObjectOutputStream* do pole bytů. Oba dva výstupy jsem pak následně ještě zkomprimoval pomocí třídy *Deflater*. Z tabulky 2 lze vidět, že použití serializace do pole bytů je z hlediska velikosti zprávy efektivnější přístup. Z tohoto důvodu jsem přistoupil k serializaci objektu třídy *Message* do pole bytů namísto původního textu formátu JSON.

Typ serializace	**Velikost po serializaci v B**	**Velikost po komprimaci v B**
JSON	19271	9369
Pole bytů	8027	7591

Tabulka 2: Porovnání velikosti zprávy při různých způsobech serializace

Třída *Master* si neustále udržuje seznam všech aktivních instancí *Slave* společně s jejich aktuálním stavem. Na základě toho stavu pak konkrétní instanci *Slave* například zasílá data nebo čeká na dokončení výpočtu. *Slave* nabývá vždy jednoho ze 4 stavů. Přehled těchto stavů zobrazuje UML stavový diagram na obrázku 24.



Obrázek 24: UML stavový diagram pro uzel typu Slave

Stav neinicializován znamená, že daný *Slave* by se měl účastnit distribuovaného výpočtu, ale zatím nenavázal žádnou komunikaci s instancí *Master*. V tomto stavu jsou na začátku všechny uzly typu *Slave* v případě manuálního módu, kdy se instanci *Master* zadává seznam instancí *Slave* pomocí konfiguračního souboru. Stav odeslat znamená, že *Slave* čeká na data od instance *Master* a *Master* mu musí zaslat aktuální verzi dat a přidělit dávku trénovacích prvků pro učení. Do tohoto stavu se může *Slave* dostat ihned na začátku a přeskočit tak stav neinicializován v případě automatického módu, kdy se instanci *Master* nezadává seznam instancí *Slave* pomocí konfiguračního souboru a konkrétní *Slave* se k instanci *Master* sám zaregistruje prostřednictvím zprávy. Stav neodeslat znamená, že *Slave* nepotřebuje žádná další data a provádí výpočet na zadané dávce vzorů. Stav ukončen znamená, že *Master* zaslal instanci *Slave* příkaz pro ukončení činnosti a *Slave* se může vypnout.

7.5.2 Implementace různých módů spouštění

Program pro paralelizaci lze spouštět ve 3 různých módech: manuální, poloautomatický a automatický. Spouštění zajišťuje třída *Starter*, která se nachází v balíčku neuronNetModeler.Aeron. Mód spouštění lze zvolit prostřednictvím parametrů příkazové řádky. V rámci této třídy jsou taky nastaveny různé vlastnosti pro knihovnu Aeron. Jedná se například o velikost bufferu pro příjem zprávy. Příkazy pro spouštění jednotlivých módu jsou uvedeny v ukázce kódu 4.

```
1 java −jar RBMDBMParallel.jar M M config.cfg data.data output.rbmw −−master manuální
2 java −jar RBMDBMParallel.jar S M 1 config.cfg data.data −−slave manuální
3 java −jar RBMDBMParallel.jar M A 10.0.0.5:41222 config.cfg data.data output.rbmw −−master
      poloautomatický
4 java −jar RBMDBMParallel.jar S A 10.0.0.5:41222 8 config.cfg data.data 41300 −−slave poloautomatický
5 java −jar RBMDBMParallel.jar A 10 config.cfg data.data output.rbmw sharedFile.txt 41300 −−automatický
```

Výpis 4: Módy spouštění programu pro paralelizaci

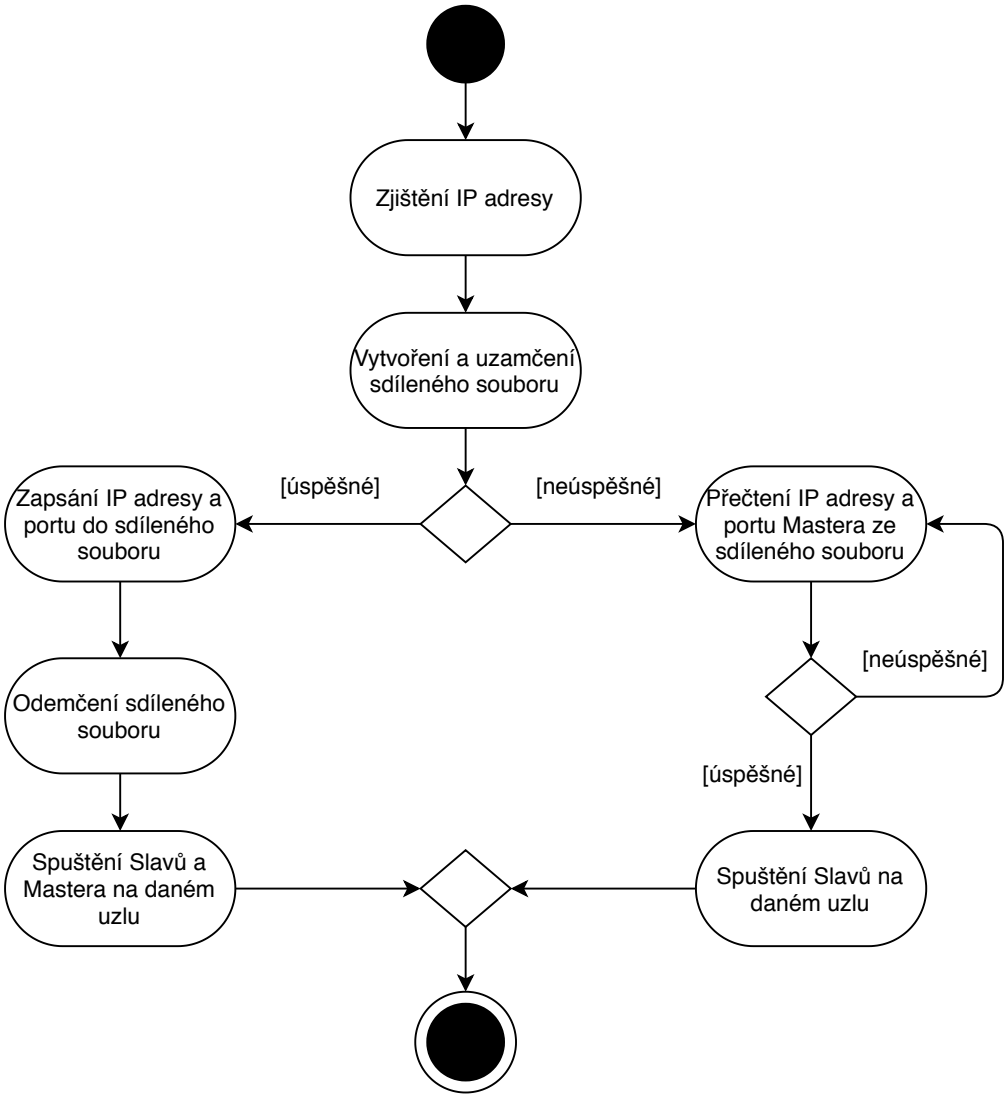
Manuální mód získává všechna data z konfiguračního souboru. Z konfiguračního souboru se načítají IP adresy a porty instancí *Slave* i IP adresa a port instance *Master*. V případě spouštění uzlu typu *Master* je první parametr písmeno M jako *Master* a druhý parametr písmeno M jako manuální. Poté následuje cesta ke konfiguračnímu souboru, k datovému souboru s trénovací množinou a k výstupnímu souboru obsahujícímu naučené váhy. V případě spouštění uzlu typu *Slave* je první parametr písmeno S jako *Slave* a druhý parametr písmeno M jako manuální. Poté následuje číslo adresy instance *Slave* v konfiguračním souboru a opět konfigurační a datový soubor.

Dalším módem je mód poloautomatický. Zde již nedochází k načítání IP adres a portů z konfiguračního souboru, ale uživatel stále určuje roli daného uzlu (*Master* nebo *Slave*). IP adresu a port instance *Master* uživatel zadává pomocí parametru příkazové řádky, ale IP adresy instancí *Slave* se zjišťují automaticky a porty se přiřazují taky automaticky. V případě spouštění uzlu typu *Master* je první parametr písmeno M jako *Master* a druhý parametr písmeno A jako automatický. Poté následuje IP adresa společně s portem a opět cesty ke všem 3 souborům.

V případě spouštění uzlu typu *Slave* je první parametr písmeno S jako *Slave* a druhý parametr písmeno A jako automatický. Poté následuje IP adresa instance *Master* společně s portem, požadovaný počet instancí typu *Slave* na daném uzlu a opět konfigurační a datový soubor. Jako poslední parametr je číslo od kterého se začnou přidělovat sekvenčně čísla portů jednotlivým instancím *Slave* na daném uzlu.

Posledním módem je mód automatický. Zde se z konfiguračního souboru načítá pouze konfigurace samotné neuronové sítě a veškeré IP adresy a porty se zjišťují automaticky. Automaticky se jednotlivé uzly také dohodnou, kdo bude *Master* a kdo *Slave*. Tento mód vyžaduje sdílené datové úložiště, ke kterému mají přístup všechny uzly. Prvním parametrem je písmeno A jako automatický. Poté následuje požadovaný počet instancí na jednom uzlu (například číslo 10 znamená 10 instancí *Slave* nebo 1 *Master* a 9 instancí *Slave*, každý spuštěný v jednom vlákně), konfigurační, datový a výstupní soubor. Za těmito soubory následuje sdílený soubor, který se vytvoří v rámci fáze rozhodování role uzlu. K tomuto sdílenému souboru musí mít přístup všechny uzly. Jako poslední parametr je číslo od kterého se začnou přidělovat sekvenčně čísla portů jednotlivým instancím na daném uzlu.

Automatický mód je naimplementován následujícím způsobem. Nejdříve si každý uzel zjistí svou IP adresu. Následně se pokusí vytvořit a uzamknout sdílený soubor na cestě uvedené v příslušném parametru příkazového řádku. Uzel, kterému se to povede se stává uzlem typu *Master* a ostatní uzly se stávají uzly typu *Slave*. *Master* do souboru následně zapíše svou IP adresu společně s portem a soubor odemkne. Následně spouští ve vláknech požadovaný počet instancí *Slave* a nakonec taky instanci třídy *Master*. Ostatní uzly si po odemknutí souboru přečtou IP adresu a port instance *Master* a ve vláknech spustí požadovaný počet instancí třídy *Slave*. Každá instance třídy *Slave* se potom pomocí zprávy zaregistruje u instance *Master* a celý proces učení může začít. Celý proces automatického módu znázorňuje aktivitní diagram na obrázku 25.



Obrázek 25: UML aktivitní diagram znázorňující automatický mód

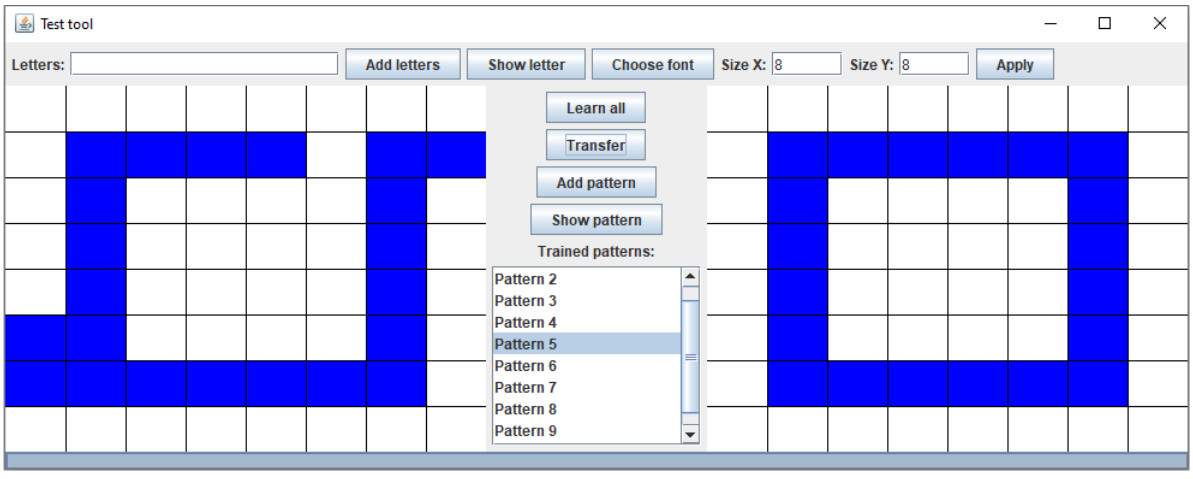
8 Uživatelská příručka

V rámci této diplomové práce jsem pracoval na dvou programech. Ani jeden z programů není třeba instalovat. Oba dva programy jsou napsány v programovacím jazyce Java, tudíž pro spuštění programů je nutné mít Javu nainstalovanou v počítači.

8.1 Modeler neuronových sítí

Po spuštění programu se zobrazí úvodní obrazovka. Pomocí zvolení možnosti *File* a následně *New...* můžete vytvořit novou neuronovou síť. Po vytvoření neuronové sítě se síť zobrazí v levé části úvodní obrazovky. Síť, kterou jsme dříve uložili pomocí tlačítka *Save* v kontextovém menu dané sítě, lze znovu otevřít pomocí voleb *File* a následně *Open...*. Program modeler neuronových sítí pro zobrazení dostupných typů neuronových sítí využívá konfigurační soubor $nnm.cfg$, který se nachází v domovském adresáři. Může se stát, že pokud jste tento program měli v minulosti v počítači spuštěný, tak neuronové sítě RBM a DBM v programu nyní neuvidíte. Tento problém lze odstranit smazáním konfiguračního souboru $nnm.cfg$ v domovském adresáři a následným opětovným spuštěním programu.

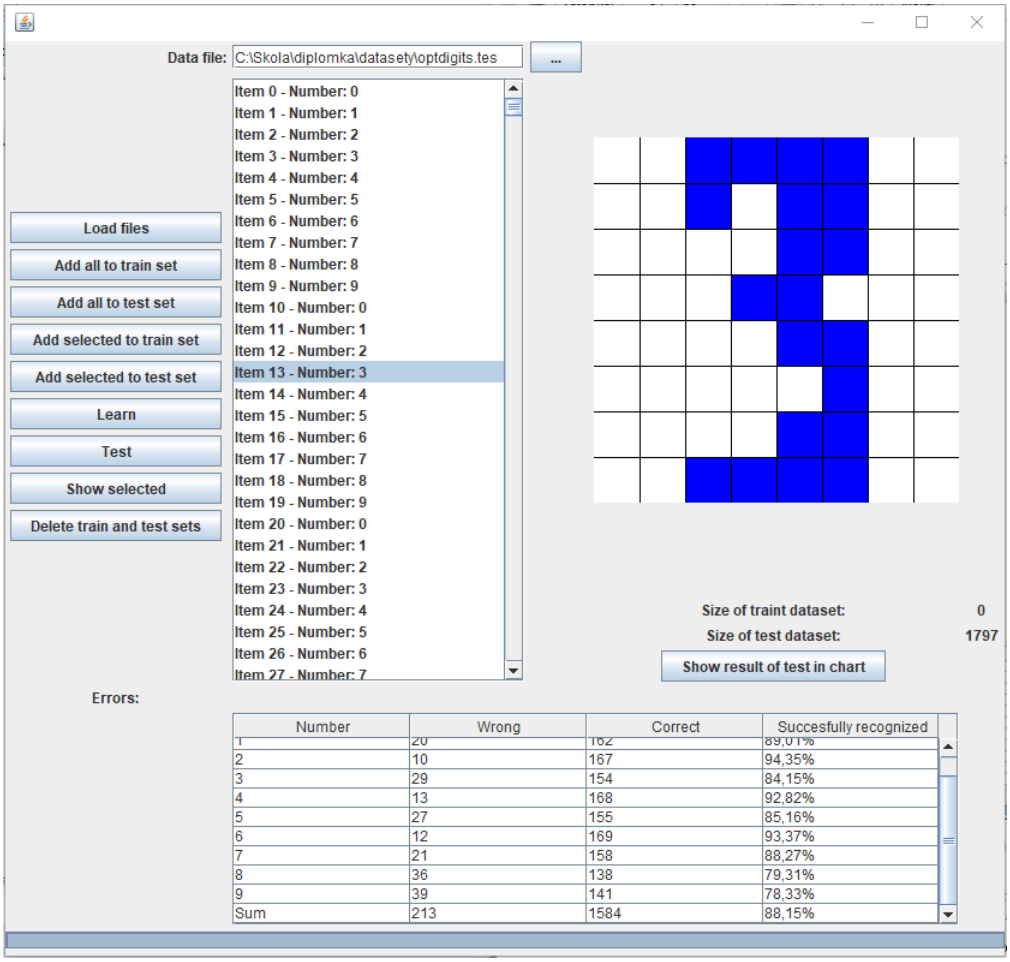
Po kliknutí pravým tlačítkem myši na vytvořenou síť se zobrazí kontextové menu. Pomocí možnosti *Testtool* zobrazíte základní nástroj pro práci s neuronovou sítí. Můžete si v něm pomocí myši vytvořit v levé části okna vlastní dataset a neuronovou síť na tento dataset naučit. Dále si pak můžete v pravé části okna zobrazit výstup neuronové sítě pro jednotlivé vstupní vzory pomocí tlačítka $Transfer$. Nástroj je zobrazen na obrázku 26.



Obrázek 26: Ukázka nástroje pro práci s neuronovou sítí

Pomocí možností pod volbou *Datasets* můžete pracovat s různými datasety. Načtený dataset lze přidat do testovací nebo do trénovací množiny u dané neuronové sítě. Pomocí tohoto nástroje lze taky spustit proces učení nad trénovací množinou a proces testování nad testovací množinou. Program umožňuje načítat také vlastní datasety ze souboru ve formátu CSV. Každý

řádek v tomto souboru reprezentuje jeden trénovací vzor. Každý element tohoto vzoru musí nabývat hodnoty 0 nebo 1 a jednotlivé elementy musí být odděleny znakem ;. Za posledním elementem na řádku se znak ; nevyskytuje. Součástí každého trénovacího vzoru jsou také elementy reprezentující jednotlivé třídy (viz. kapitola 3.4 Rozšíření RBM pro klasifikaci). Ukázka nástroje pro práci s datasety je zobrazena na obrázku 27.



Obrázek 27: Ukázka nástroje pro práci s datasety

Pod volbou *Monitoring tools* v kontextovém menu dané sítě se nachází jednotlivé monitorovací nástroje uvedené v kapitole 5 Monitorování učení neuronové sítě. Každý nástroj je nutné před zahájením procesu učení spustit a zaregistrovat k neuronové síti pomocí tlačítka *Start*. Okno s tímto nástrojem se nesmí během procesu učení zavřít. U každého nástroje lze nastavit jak často se má daná vizualizace přepočítávat a u některých nástrojů lze zvolit zda se má počítat z trénovací nebo z testovací množiny.

Pomocí volby *Parallel learning* v kontextovém menu dané sítě se nachází nástroj pro vytváření souborů pro program pro distribuovaný výpočet. Lze zde uložit datový soubor obsahující trénovací vzory přiřazené u zvolené neuronové sítě. V tomto okně se nastavuje velikost dávky

pro paralelní učení a počet epoch vykonaných v rámci jedné dávky. Také zde lze nastavit IP adresy a porty jednotlivých uzlů pro použití manuálního módu programu. Pomocí tlačítka *Save configuration file* lze uložit konfigurační soubor obsahující nastavení neuronové sítě i nastavení paralelizace provedené v tomto okně.

Pomocí tlačítka *Import weights* v kontextovém menu zvolené sítě lze do sítě vložit uložené váhy vzniklé během paralelního učení. Vytvořená neuronová síť musí mít počty neuronů ve všech vrstvách shodné s těmi, co jsou v souboru vah, který chceme do sítě vložit.

8.2 Program pro distribuovaný výpočet

Jedná se o konzolovou aplikaci, která jako vstup vyžaduje konfigurační a datový soubor vytvořený v Modeleru neuronových sítí. Program lze spouštět v různých módech. Jednotlivé módy společně s přehledem jejich parametrů jsou uvedeny v kapitole 7.5.2 Implementace různých módů spouštění. Pro automatický mód je nutné mít sdílené úložiště ke kterému mají přístup všechny uzly.

Program vytváří na uzlu typu $Master$ log soubor s názvem $log.txt$. Z obsahu tohoto souboru můžete zjistit aktuální stav procesu učení. Naleznete zde čas začátku a konce výpočtu, zbývající počet epoch a počet zaregistrovaných instancí $Slave$ účastnících se výpočtu. Dále zde také najdete IP adresy a porty všech instancí $Slave$ a instance $Master$. V tomto souboru naleznete taky případné chyby, které nastaly během vykonávání programu.

Výstupem tohoto programu je soubor obsahující naučené váhy. Tento soubor naleznete po ukončení procesu učení na cestě, kterou jste uvedli v daném parametru příkazové řádky. Vytvořený soubor lze následně importovat do neuronové sítě v programu Modeler neuronových sítí.

9 Testování a experimenty

Tato kapitola se zabývá testováním neuronových sítí RBM a DBM. Cílem tohoto testování bylo ověřit naimplementovanou funkčnost neuronových sítí a zjistit, jaký vliv má datový paralelismus na kvalitu a rychlost učení těchto neuronových sítí.

9.1 Testování sekvenčního učení

Cílem tohoto testování nebylo nalézt co možná nejlepší parametry neuronových sítí pro dosažení co možná nejvyšší úspěšnosti při klasifikaci. V tomto testování jsem se zaměřil na dva hlavní cíle. Prvním cílem bylo ověřit funkčnost naimplementovaných neuronových sítí RBM a DBM. Druhým cílem bylo zjistit rychlost a kvalitu učení na zvolených datasetech. Výsledky tohoto testování budou následně použity pro zjištění toho, jak velký vliv má datový paralelismus na rychlost a kvalitu.

Nejdříve bylo provedeno testování v lokálním prostředí. Cílem tohoto testování bylo na velmi jednoduché a malé trénovací sadě dat odhalit případné chyby v implementaci. Malá a jednoduchá trénovací množina byla zvolena proto, aby se malé neuronové sítě dokázaly tyto trénovací data velmi rychle naučit. Po odstranění chyb a úspěšném učení na této malé trénovací množině mohlo začít testování v serverovém prostředí.

Cílem testování v serverovém prostředí bylo zjistit rychlost a kvalitu učení na zvolených datasetech. Testování bylo prováděno na virtuálních serverech vytvořených univerzitou. Výhodou bylo, že běžely nepřetržitě a učení tak mohlo probíhat i velmi dlouhou dobu. Konfigurace serveru je uvedena v tabulce 3. Výsledky tohoto testování jsou popsány v následujících dvou podkapitolách.

Název	Virtuální server
OS	Ubuntu 18.04.3 LTS
CPU	QEMU Virtual CPU 2GHz
RAM	4GB

Tabulka 3: Konfigurace serverového testovacího prostředí

9.1.1 Restricted Boltzmann Machine

Nejvíce experimentů bylo provedeno na datasetu OPT digits. Tento dataset je poměrně malý a jednoduchý a proto netrvalo učení příliš dlouhou dobu. První sadu experimentů jsem provedl s následujícími parametry:

- Počet viditelných neuronů 74
- Počet skrytých neuronů 200
- Momentum 0,1

- Učící faktor 0,01
- Velikost dávky 1

Během experimentů jsem postupně zvyšoval počet epoch a sledoval, jaký vliv má počet epoch na rychlost a kvalitu učení neuronové sítě. Výsledky těchto experimentů zobrazuje tabulka 4. Zajímavé je, že již po první epoše má neuronová síť poměrně velkou úspěšnost a tato úspěšnost se s přibývajícím počtem epoch zvyšuje velmi pomalu. Rozdíl v úspěšnosti při počtu epoch 50 a 100 je již téměř neznatelný.

Počet epoch	**Čas v s**	**Úspěšnost v %**
1	1	76,13
2	3	81,19
4	6	83,03
6	9	84,92
8	12	85,09
10	16	86,31
30	48	86,64
50	79	88,15
100	158	88,17
150	237	88,17

Tabulka 4: Vliv počtu epoch na rychlost a úspěšnost učení sítě RBM

Druhá sada experimentů probíhala na stejném datasetu a se stejnými parametry. Na základě předchozí sady experimentů jsem zvolil pevný počet epoch 50 a tentokrát jsem vyzkoušel proměnný počet skrytých neuronů. Výsledky těchto experimentů zobrazuje tabulka 5. Již při 100 skrytých neuronech, což je desetinásobek počtu tříd tohoto datasetu, dosahuje neuronová síť poměrně dobré úspěšnosti.

Počet skrytých neuronů	**Čas v s**	**Úspěšnost v %**
5	3	63,55
10	4	77,74
30	9	84,14
50	17	85,70
100	37	87,59
150	59	88,05
200	79	88,15
250	106	87,53

Tabulka 5: Vliv počtu skrytých neuronů na rychlost a úspěšnost učení sítě RBM

Třetí sada experimentů již probíhala na různých datasetech. Na základě předchozích experimentů jsem zvolil 50 epoch. Počet skrytých neuronů jsem zvolil jako desetinásobek počtu tříd v daném datasetu. Výsledky těchto experimentů zobrazuje tabulka 6. Neuronová síť RBM při těchto parametrech dosahuje poměrně dobrých výsledků i v případě komplikovaného datasetu Caltech 101 Silhouettes. Tyto výsledky mohou následně posloužit pro porovnání při testování paralelního učení sítě RBM.

Dataset	**Viditelné neurony**	**Skryté neurony**	**Čas**	**Úspěšnost v %**
OPT	74	100	37s	87,59
MNIST	794	100	3h53m35s	85,96
OCR	154	260	46m18s	66,09
Caltech28	885	1000	6h1m51s	62,33

Tabulka 6: Sekvenční testování učení sítě RBM na různých datasetech

9.1.2 Deep Boltzmann Machine

Učení této neuronové sítě již trvá oproti síti RBM mnohem déle. Při nastavení parametrů jsem se rozhodl vyjít z předchozích experimentů sítě RBM, jelikož síť DBM je z několika těchto sítí RBM složená. Pro experimenty jsem použil síť se dvěma skrytými vrstvami, kdy první skrytá vrstva má nastaven počet neuronů jako desetinásobek počtu tříd v daném datasetu a druhá skrytá vrstva jako dvojnásobek počtu neuronů první skryté vrstvy. Počet epoch jsem v obou fázích procesu učení zvolil 50 a učící faktor v globální fázi učení jsem oproti fázi předtrénování snížil.

Přehled použitých parametrů je uveden v tabulce 7 a výsledky experimentů zobrazuje tabulka 8. V případě datasetu OPT digits byl použit počet vzorkovacích částic 100. Tyto výsledky mohou následně posloužit pro porovnání při testování paralelního učení sítě DBM. Neuronová síť DBM při těchto parametrech dosahuje přibližně stejných výsledků jako RBM.

Učící faktor - fáze předtrénink	0,01
Učící faktor - fáze trénink	0,001
Momentum - obě fáze	0,1
Počet epoch - obě fáze	50
Počet vzorkovacích částic	200
Koeficient snižování učícího faktoru	0,99
Počet kroků Gibbsova vzorkování	5
Počet Mean-field kroků	50

Tabulka 7: Parametry neuronové sítě DBM

Dataset	Viditelné neurony	Skryté neurony vrstva 1 / vrstva 2	Čas	Úspěšnost v %
OPT	74	100/200	43m35s	85,59
MNIST	794	100/200	4d17h26m49s	90,39
OCR	154	260/520	7d2h23m39s	66,87

Tabulka 8: Sekvenční testování učení sítě DBM na různých datasetech

9.2 Testování paralelního učení

V tomto testování jsme se zaměřili na to, jak datový paralelismus ovlivní rychlost učení a úspěšnost klasifikace neuronových sítí RBM a DBM. Nejdříve bylo provedeno testování v lokálním prostředí. Cílem tohoto testování bylo na velmi jednoduché a malé trénovací sadě dat odhalit případné chyby v implementaci. Po odstranění chyb a úspěšném učení na této malé trénovací množině začalo testování v serverovém prostředí. Úkolem tohoto testování bylo ověřit, zda bude síťová komunikace mezi uzly probíhat správně i přesto, že uzly nebudou spuštěny v rámci jednoho stroje. Po úspěšném učení v serverovém prostředí mohlo začít testování na superpočítači. Testování probíhalo na superpočítači Salomon. Výsledky tohoto testování jsou popsány v následujících dvou podkapitolách. Superpočítač Salomon [2] se skládá z 1008 uzlů a využívá počítačové sítě InfiniBand FDR56. Konfigurace jednoho uzlu je uvedena v tabulce 9.

OS	CentOS 7.x Linux
CPU	2 x Intel Xeon E5-2680v3, 2.5 GHz, 12 jader
RAM	128GB, 5.3 GB na jádro, DDR4@2133 MHz

Tabulka 9: Konfigurace uzlu superpočítače Salomon

Abychom mohli program na superpočítači spustit, musíme vytvořit následující skript startBatch uvedený v ukázce kódu 6 a startProgram, který může vypadat například tak, jak je uvedeno v ukázce kódu 5. Tento skript spustí instanci programu na všech superpočítačem přidělených uzlech. Následně již můžeme požadavek na výpočet přidat do fronty například pomocí příkazu uvedeného v ukázce kódu 7. V rámci tohoto příkazu musíme zadat kód přiděleného projektu, typ fronty (qprod znamená produkční fronta), počet uzlů a procesorů a název skriptu.

```
1 #!/bin/bash
2 module add Java/1.8.0_202
3 java -jar RBMDBMParallel.jar A 10 configRBM.cfg dataMNIST.data resultMNIST.rbmw masterAddress.txt
      40450
```

Výpis 5: Skript startProgram

```
#!/bin/bash
lines=($(sort -u $PBS_NODEFILE))
h=${lines[0]}
for (( i=1; i<${#lines[@]};i++));
do
h=$h,${lines[i]}
done
pdsh -R exec -w $h ssh %h ./startProgram
exit
```

Výpis 6: Skript startBatch

```
qsub -A kod_projektu -q qprod -l select=2:ncpus=24 ./startBatch
```

Výpis 7: Příkaz pro přidání požadavku na výpočet do fronty

9.2.1 Restricted Boltzmann Machine

Nejvíce experimentů bylo opět provedeno na datasetu OPT digits, protože tento dataset je poměrně malý a učení netrvá příliš dlouhou dobu. Parametry jsem zvolil stejné jako jsou uvedené v podkapitole 9.1.1 Restricted Boltzmann Machine. Počet epoch jsem zvolil 150 a počet skrytých neuronů 200. Tyto o něco vyšší hodnoty jsem zvolil z toho důvodu, aby učení netrvalo příliš krátkou dobu.

V první sadě experimentů jsem vyzkoušel měnit velikost dávky při pevném počtu paralelních instancí typu Slave. Zvolil jsem počet instancí typu Slave 9, kdy všechny instance byly na jednom uzlu, jelikož jeden uzel superpočítače má 24 procesorových jader. Výsledek tohoto experimentu je zobrazen v tabulce 10. V případě sekvenčního učení se úspěšnost při těchto parametrech pohybovala kolem 88%. Z tabulky lze vidět, že úspěšnost neuronové sítě s rostoucí velikostí dávky bohužel prudce klesá a při velikosti dávky 250 se již úspěšnost rozpoznání rovná náhodnému výběru. Naopak snižovat velikost dávky pod 25 nemá smysl, protože režie vzniklá při zasílání zpráv při paralelizaci by byla již příliš velká a nedosáhli bychom žádného časového zrychlení. Z počátku klesá čas velmi rychle a od velikosti dávky 150 již velmi pomalu.

Velikost dávky	**Čas v s**	**Úspěšnost v %**
25	175	79,13
50	95	71,12
100	61	57,54
150	55	36,00
200	50	19,81
250	48	9,91

Tabulka 10: Vliv velikosti dávky na rychlost a úspěšnost učení sítě RBM

Ve druhé sadě experimentů jsem vyzkoušel měnit počet paralelních instancí typu Slave při pevné velikosti dávky. Velikost dávky jsem zvolil 50, protože při této velikosti v předchozí sadě experimentů nebyl pokles úspěšnosti příliš velký. Při počtu instancí Slave nižším než 10 běží všechny instance na jednom uzlu, při vyšším počtu jsou instance rovnoměrně rozděleny mezi dva uzly. Výsledek tohoto experimentu je zobrazen v tabulce 11. Z tabulky lze vidět, že bohužel opět dochází k poměrně prudkému poklesu úspěšnosti. Paralelizace při této velikosti dávky se navíc při třech instancích typu Slave nevyplatí, protože při tomto počtu instancí trvá paralelní výpočet déle než výpočet sekvenční. Paralelizace se při této velikosti dávky vyplatí až při vyšším počtu instancí typu Slave. Toto je způsobeno tím, že přenos zpráv a synchronizace vah trvá nějaký čas, a při tak malé velikosti dávky musí k přenosu zpráv docházet velmi často. Příliš velký počet instancí typu Slave při této velikosti dávky ovšem také nemá smysl. Z tabulky lze vidět, že při počtu instancí 15 a 19 je čas srovnatelný. To je způsobeno tím, že Master při tak malé velikosti dávky již nestíhá přijaté delty vah od instancí Slave integrovat do své globální verze a zadávat jim nové požadavky na výpočet. Master se pak stává úzkým místem a brzdí všechny ostatní instance Slave. Toto je způsobeno tím, že instancím Slave trvá výpočet takto malé dávky příliš krátkou dobu.

Počet instancí Slave	**Čas v s**	**Úspěšnost v %**
3	280	81,69
6	134	72,68
9	95	71,12
11	110	66,33
15	95	66,11
19	97	49,69

Tabulka 11: Vliv počtu instancí Slave na rychlost a úspěšnost učení sítě RBM

Předchozí dvě sady experimentů ukázaly, že dosažení dobré úspěšnosti a rychlosti při paralelním učení není možné dosáhnout změnou velikosti dávky ani změnou počtu instancí Slave. V další sadě experimentů jsem se pokusil provádět učení přidělené dávky po více epoch a až poté vrátit delty vah instanci Master. Tyto experimenty jsem prováděl při velikosti dávky 50 a počtu instancí Slave 9. Výsledek tohoto experimentu zobrazuje tabulka 12. Při použití vyššího počtu epoch v rámci jedné dávky sice došlo ke zrychlení, ale úspěšnost okamžitě klesla na úroveň, kdy se téměř rovnala náhodnému výběru.

Počet epoch v dávce	**Čas v s**	**Úspěšnost v %**
1	95	71,12
3	53	19,87
5	51	10,07

Tabulka 12: Vliv počtu epoch v rámci jedné dávky na rychlost a úspěšnost učení sítě RBM

V dalších experimentech jsem se problémy s úspěšností při paralelním učení snažil vyřešit různými způsoby. Zkoušel jsem například zvyšovat celkový počet epoch učení. Toto řešení ale vedlo pouze ke zvýšení času učení a na zvýšení úspěšnosti nemělo žádný vliv. Také jsem vyzkoušel během učení na přidělené dávce v instanci Slave neaktualizovat průběžně váhy. Tedy daný Slave prochází postupně všechny mu přidělené učící vzory v dávce a ukládá si vypočtené delty vah (pro odeslání instanci Master), ale tyto delty vah po každém vzoru neaplikuje do své lokální verze vah. Lokální verze vah tak zůstává pro konkrétní instanci Slave stejná po celou dobu učení přidělené dávky a mění se pouze zasláním nové verze vah od instance Master při přidělení nové dávky. Bohužel i tento přístup nepřinesl žádný výsledek, protože úspěšnost v tomto případě při velikosti dávky 50 a počtu instancí Slave 9 klesla na úroveň, kdy se téměř rovnala náhodnému výběru.

Dataset OPT digits obsahuje 3823 trénovacích vzorů. Při počtu instancí Slave 9 a velikosti dávky 50 se tak paralelně zpracovává 450 vzorů, což je přibližně 11% trénovací množiny. V dalším experimentu jsem tak vyzkoušel použít při stejném počtu instancí Slave dataset MNIST. Při experimentu jsem zvolil 50 epoch a stejné parametry sítě jako v případě sekvenčního testování tohoto datasetu. Dataset MNIST obsahuje 60 000 trénovacích vzorů. Při velikosti dávky 50 se paralelně zpracovává 450 vzorů, což je méně než 1% trénovací množiny, a při velikosti dávky 100 se paralelně zpracovává 1,5% trénovací množiny. Výsledky tohoto experimentu zobrazuje tabulka 13. Sice došlo k velkému zrychlení učení, ale opět došlo k významnému poklesu úspěšnosti, protože úspěšnost při sekvenčním učení byla téměř 86%. Vypadá to tedy, že velikost trénovací množiny vzhledem k velikosti dávky nemá u této neuronové sítě na úspěšnost vliv.

Velikost dávky	**Čas**	**Úspěšnost v %**
50	40m1s	63,35
100	21m54s	48,20

Tabulka 13: Vliv velikosti dávky na rychlost a úspěšnost učení sítě RBM - dataset MNIST

Z předchozích experimentů je patrné, že u neuronové sítě RBM dochází při použití datového paralelismu velmi rychle k úpadku úspěšnosti. Velikost dávky by musela být příliš malá, a to se naopak nevyplatí z důvodu rychlosti, protože synchronizace vah mezi jednotlivými dávkami prostřednictvím zasílání zpráv v distribuovaném výpočtu představuje značnou režii. V experimentech byla dokonce využita velmi malá část výkonu superpočítače, protože experimenty ukázaly, že větší počet instancí typu Slave nemá smysl uvažovat. Tato neuronová síť, v případě použití datového paralelismu, tedy není vhodná pro paralelní distribuovaný výpočet. V budoucnu by možná stálo za pokus vyzkoušet pouhou vícevláknovou aplikaci na jednom stroji, kdy by odpadla režie se zasíláním zpráv, a mohli bychom si tak dovolit velmi malé dávky a malý počet paralelně běžících výpočtů. Možná by se tak dosáhlo přijatelného zrychlení při přijatelném poklesu úspěšnosti.

9.2.2 Deep Boltzmann Machine

Většinu experimentů jsem opět prováděl na datasetu OPT digits. Parametry jsem zvolil stejné jako jsou uvedeny v podkapitole 9.1.2 Deep Boltzmann Machine. V první sadě experimentů jsem opět vyzkoušel měnit velikost dávky při pevném počtu paralelních instancí typu Slave. Zvolil jsem počet instancí typu Slave 9, kdy všechny instance byly na jednom uzlu. Výsledek tohoto experimentu je zobrazen v tabulce 14. Z tabulky lze vidět, že již při velikosti dávky 25 úspěšnost prudce klesla. S rostoucí velikostí dávky úspěšnost již klesá relativně pomalu. Dále lze z tabulky vidět, že u této sítě již při velikosti dávky 25 došlo k většímu zrychlení než v případě sítě RBM. Toto je způsobeno tím, že samotný výpočet v případě této sítě trvá delší dobu a režie spojená se zasíláním zpráv tak představuje menší část celkového času paralelního učení než v případě sítě RBM.

Velikost dávky	**Čas**	**Úspěšnost v %**
25	10m15s	42,18
50	7m1s	42,18
100	5m38s	41,01
150	5m14s	38,95
200	4m57s	36,00
250	4m47s	32,11

Tabulka 14: Vliv velikosti dávky na rychlost a úspěšnost učení sítě DBM

Ve druhé sadě experimentů jsem vyzkoušel měnit počet paralelních instancí typu Slave při pevné velikosti dávky. Velikost dávky jsem zvolil 50. Při počtu instancí Slave nižším než 10 běží všechny instance na jednom uzlu, při vyšším počtu jsou instance rovnoměrně rozděleny mezi dva uzly. Výsledek tohoto experimentu je zobrazen v tabulce 15. Z tabulky lze vidět, že opět již při malém počtu instancí typu Slave bohužel dochází k velkého poklesu úspěšnosti. Při zvyšujícím se počtu instancí Slave již úspěšnost klesá poměrně pomalu. Při zvyšujícím se počtu instancí Slave čas běhu paralelního výpočtu stále klesá.

Počet instancí Slave	**Čas**	**Úspěšnost v %**
3	19m41s	51,86
6	9m47s	50,19
9	7m1s	42,18
11	6m17s	48,08
15	5m16s	40,90
19	4m34s	37,34

Tabulka 15: Vliv počtu instancí Slave na rychlost a úspěšnost učení sítě DBM

Dále jsem se pokusil provést experimenty na větším datasetu MNIST. Vyzkoušel jsem velikosti dávek 50 a 100 při počtu instancí Slave 9. Opět sice došlo k výraznému zrychlení učení, kdy například učení při velikosti dávky 100 probíhalo kolem 10 hodin, ale bohužel v případě těchto experimentů se úspěšnost téměř rovnala náhodnému výběru.

Z předchozích experimentů vyplývá, že u neuronové sítě DBM dochází při použití datového paralelismu již u použití malé dávky a malého počtu instancí typu Slave k velkému poklesu úspěšnosti. Vypadá to, že použití datového paralelismu není pro tento typ neuronové sítě vhodné.

10 Závěr

Neuronové sítě můžeme využít pro řešení spousty různých úloh. V této práci jsem se zabýval neuronovými sítěmi Restricted Boltzmann Machine a Deep Boltzmann Machine. O obě dvě tyto sítě jsem rozšířil program Modeler neuronových sítí. Program jsem také rozšířil o možnost monitorování průběhu jejich učení a o různé nástroje pro práci s datasety. Nevýhodou neuronových sítí je, že jejich učení obvykle vyžaduje velký výpočetní výkon a trvá velmi dlouhou dobu. Jednou z možností, jak velkého výkonu a kratší doby učení dosáhnout je pomocí využití paralelizace učení. V rámci této diplomové práce jsem se zabýval paralelizací na více strojích (uzlech). Vytvořil jsem program, který funguje na principu datového paralelismu v kombinaci s asynchronním centralizovaným přístupem a komunikačním protokolem UDP. Jako základ tohoto programu jsem použil knihovnu vzniklou v rámci diplomové práce Ing. Vojtěcha Pustówky.

S naimplementovaným řešením jsem prováděl spoustu různých experimentů jak na klasickém počítači, tak na superpočítači. Cílem těchto experimentů bylo zjistit, jak velký vliv bude mít použitý způsob paralelizace na rychlost učení a úspěšnost rozpoznávání těchto neuronových sítí, protože tento způsob paralelizace byl již úspěšně použit pro sítě s metodou učení Backpropagation.

Z provedených experimentů lze vidět, že u neuronových sítí RBM a DBM dochází při použití datového paralelismu sice ke zrychlení učení, ale také velmi rychle k velkému poklesu úspěšnosti. V experimentech jsem vyzkoušel několik různých přístupů (změna velikosti dávky, změna počtu instancí typu Slave, navýšení počtu epoch učení a další), ale bohužel ani jeden z nich nevedl k uspokojivým výsledkům. Velikost dávky by musela být příliš malá, a to se nevyplatí z důvodu rychlosti, protože synchronizace vah mezi jednotlivými dávkami prostřednictvím zasílání zpráv v distribuovaném výpočtu představuje značnou režii. Navíc v případě sítě DBM dochází při použití datového paralelismu již u použití malé dávky a malého počtu instancí typu Slave k velkému poklesu úspěšnosti.

V případě sítě RBM a velmi malé dávky a malém počtu instancí typu Slave nebyl pokles úspěšnosti příliš velký. V budoucnu by možná stálo za pokus vyzkoušet pouhou vícevláknovou aplikaci na jednom stroji, kdy by odpadla režie se zasíláním zpráv, a mohli bychom si tak dovolit velmi malé dávky a malý počet paralelně běžících výpočtů. Možná by se tak dosáhlo přijatelného zrychlení při přijatelném poklesu úspěšnosti.

Literatura

[1] Aeron. `github.com/real-logic/aeron`. Dostupné: 5.5.2020.

[2] It4innovations documentation. `https://docs.it4i.cz/`. Dostupné: 5.5.2020.

[3] Java. `www.java.com`. Dostupné: 5.5.2020.

[4] Java - dnes při šálku dobré kávy. `www.linuxexpres.cz/praxe/java-dnes-pri-salku-dobre-kavy`. Dostupné: 5.5.2020.

[5] Jfreechart. `www.jfree.org/jfreechart/`. Dostupné: 5.5.2020.

[6] Maven. `http://maven.apache.org/`. Dostupné: 5.5.2020.

[7] Mnist. `yann.lecun.com/exdb/mnist/`. Dostupné: 5.5.2020.

[8] Ocr letters. `ai.stanford.edu/~btaskar/ocr/`. Dostupné: 5.5.2020.

[9] Opt digits. `archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits`. Dostupné: 5.5.2020.

[10] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[11] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[12] Scott Chacon. Pro git. *CZ.NIC*, 2009.

[13] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.

[14] Ray J Frank, Neil Davey, and Stephen P Hunt. Time series prediction and neural networks. *Journal of intelligent and robotic systems*, 31(1-3):91–103, 2001.

[15] Simon S Haykin et al. *Neural networks and learning machines/Simon Haykin.* New York: Prentice Hall„ 2009.

[16] Geoffrey E Hinton. A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*, pages 599–619. Springer, 2012.

[17] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543. ACM, 2008.

[18] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[19] Camelia-Mihaela Pintea. Bio-inspired computing. In *Advances in Bio-inspired Computing for Combinatorial Optimization Problems*, pages 3–19. Springer, 2014.

[20] Vojtěch Pustówka. Rozšiřující knihovna pro modeler neuronovỳch sítí. 2019.

[21] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[22] Ruslan Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015.

[23] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial intelligence and statistics*, pages 448–455, 2009.

[24] Ruslan Salakhutdinov and Hugo Larochelle. Efficient learning of deep boltzmann machines. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 693–700, 2010.

[25] Ivo VONDRÁK. Neuronové sítě. *Fakulta elektrotechniky a informatiky VŠB. Učební text. Ostrava*, 2009.

[26] Jason Yosinski and Hod Lipson. Visually debugging restricted boltzmann machine training with a 3d example. In *Representation Learning Workshop, 29th International Conference on Machine Learning*, 2012.